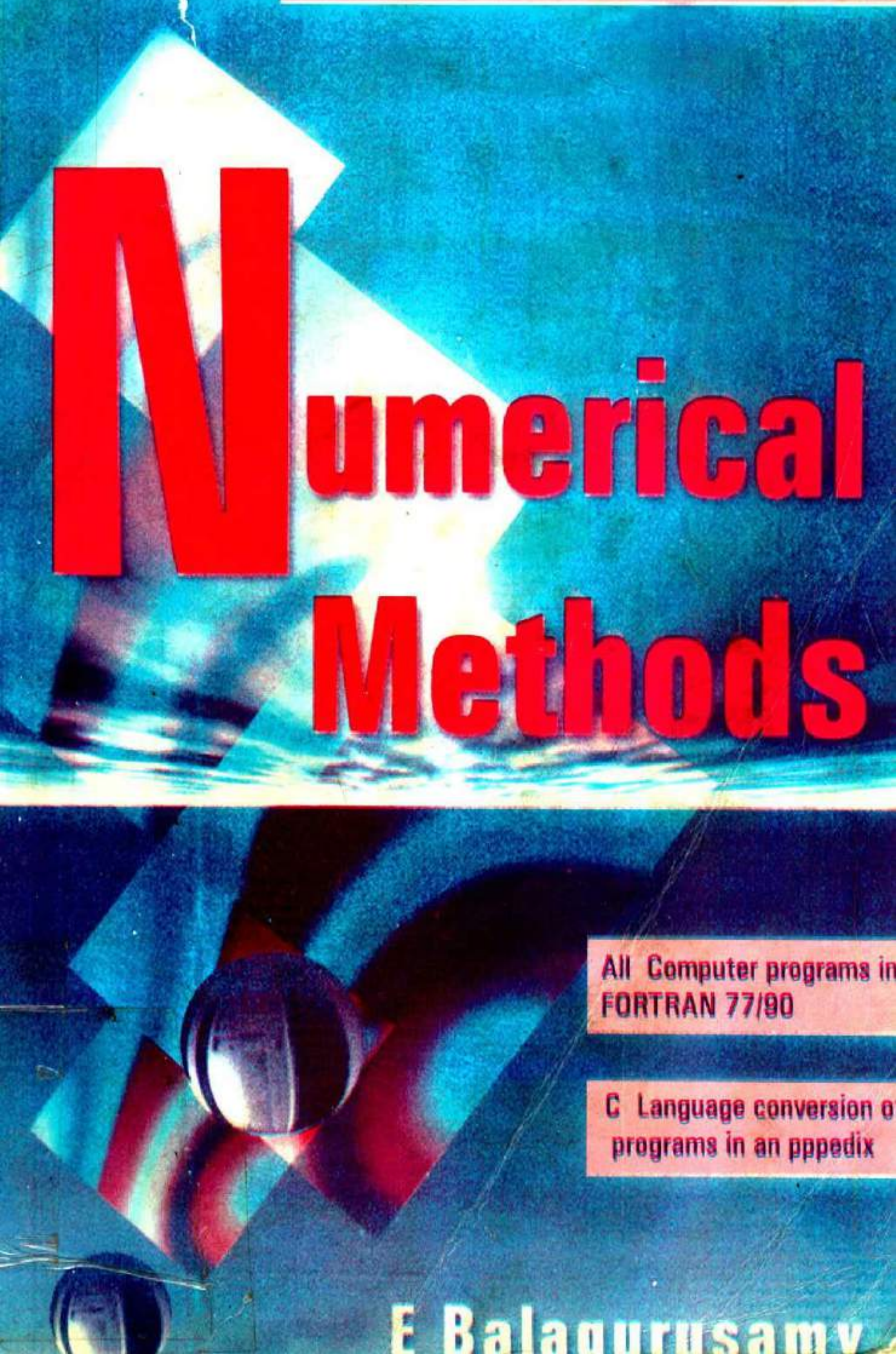
*Also available*→ a diskette version with FORTRAN and C languages source codes



# Numerical Methods

All Computer programs in FORTRAN 77/90

C Language conversion o programs in an pppedix

E Balagurusamy

# Numerical Methods

## ABOUT THE AUTHOR

**E Balagurusamy** holds an ME (Hons) degree in Electrical Engineering and a Ph D in Systems Engineering. He has been a teacher, trainer, and consultant in the area of computing for the past two decades. He has also served as a consultant to the United Nations Development Organisation (UNIDO) for a number of years. Currently, he is the Director of PSG Institute of Management, Coimbatore, a premier center for management sciences in the country.

A prolific writer, Prof. Balagurusamy has published a large number of research papers in international journals of repute like IEEE Transactions on Reliability, International Journal of Mathematics, Science and Education and International Journal of Systems Science. He has also authored several books including *Reliability Engineering* (TMH) and *Computer-Oriented Statistical and Numerical Methods*. His best selling books on computing include *Programming in ANSI C, Object- Oriented Programming with C++, Programming with Java*, and *FORTRAN for Beginners* (all by TMH).

A recipient of numerous honours and awards, he has been listed in the Directory of Who's Who of Intellectuals and in the Directory of Distinguished Leaders in Education.

# Numerical Methods

**E Balagurusamy**
*Director*
*PSG Institute of Management*
*Coimbatore*

2004

**Tata McGraw-Hill Publishing Company Limited**
NEW DELHI

*McGraw-Hill Offices*
**New Delhi** New York St Louis San Francisco Auckland Bogotá Caracas
Lisbon London Madrid Mexico City Milan Montreal
San Juan Singapore Sydney Tokyo Toronto

*Tata McGraw-Hill*

*A Division of The* ***McGraw-Hill*** *Companies*

Information contained in this work has been obtained by Tata McGraw-Hill Publishing Company Limited, from sources believed to be reliable. However, neither Tata McGraw-Hill nor its authors guarantee the accuracy or completeness of any information published herein, and neither Tata McGraw-Hill nor its authors shall be responsible for any errors, omissions, or damages arising out of use of this information. This work is published with the understanding that Tata McGraw-Hill and its authors are supplying information but are not attempting to render engineering or other professional services. If such services are required, the assistance of an appropriate professional should be sought

Fourth reprint 2000
**RQDCRRYCRZZDC**
Reprint-2004

**ISBN 0-07-463311-2**

Published by Tata McGraw-Hill Publishing Company Limited,
7 West Patel Nagar, New Delhi 110 008, typeset in New Century at
The Composers, 20/5 Old Market, West Patel Nagar, New Delhi 110 008,
and printed at Rashtriya Printers, Naveen Shahdara, Delhi 110 032

Cover: Rashtriya

# Preface

Application of computer-oriented *numerical methods* has become an integral part of the life of all the modern engineers and scientists. The advent of powerful small computers and workstations has tremendously increased the speed, power and flexibility of numerical computing. Recognizing the importance of computers and numerical computing, all the universities now offer courses in both computing and numerical methods in their engineering curriculum. They are also trying to integrate computer-oriented numerical analysis into other courses such as mathematics and computer science. It is believed that students who can understand, enjoy, and successfully apply the methods of numerical computing to solve problems make better logicians and thereby better engineers and scientists.

The contents of this book are derived from a study of curricula offered by various universities across the country. The book covers both the introductory topics and the more advanced topics such as eigenvalue problems and partial differential equations. The primary goal is to provide students with a sound introduction of numerical methods as well as to make the learning a pleasurable experience. Logical arrangement of topics, clarity of presentation, and illustration through examples aid the student to become more and more adept in applying the methods.

There are a number of books in this field. Why another book? This book is uniquely different from many other books in a number of ways. The salient features are:

- Presents a detailed account of *process* and *characteristics* of numerical computing
- Discusses the *concept of computing* using modern computers.
- Places emphasis on the nature of *computer arithmetic* which is different from real arithmetic in a number of ways.
- The concept, cause and consequence of *errors* in the application of numerical computing have been highlighted.

- An overview of FORTRAN 77/90 has been given which would serve as an introduction to the beginners and as a reference to those who have already learned this language.
- Mathematical derivation of each method is given to build the reader's understanding of numerical analysis.
- Wherever possible, *algorithms* of computing are given in pseudocode using boxes.
- A variety of *solved examples* that would sharpen the skills in both the theory and application of numerical computing are provided.
- *Computer programs* for almost all numerical methods discussed have been presented in both FORTRAN 77 and ANSI C languages.
- A separate diskette version of the book is available; the diskette contains the source codes of programs in FORTRAN 77 and C languages.
- Programs are designed using the modern *modular and structured programming* concepts. These modules can be used as building blocks in other programs.
- *Error analysis* of each method is presented.
- Each chapter begins with a statement of *need and scope* giving a preview of what is coming later.
- Each chapter ends with a *summary* and a set of *key terms* that remind the reader what has been covered and discussed.
- *Review questions* provide an opportunity to the reader to test his understanding of the concepts.
- Numerous end-of-chapter *practice exercises* are an important supplement to the text. They would not only improve the understanding of algorithms but also enhance the application skills.
- *Programming projects* give students ample opportunities to practice their skills of scientific programming.

I have exerted a conscious effort to make the book *student-oriented* and *student-friendly*. I hope the students would find the book not only interesting but also useful.

I wish to acknowledge the many helpful suggestions of reviewers which have certainly improved both the content and quality of the material. It is the persuasion and encouragement of Dr N Subrahmanyam and Vibha Mahajan of Tata McGraw-Hill that has made the publication of this book possible. Thanks are due to K Balakrishnan and J R Pratibha whose excellent word processing skills made the preparation of the book in its present form much easier and possible.

The book is full of equations and formulas that contain a large number of variables, subscripts, superscripts, etc. I shall be grateful, if the readers could communicate to me any errors they discover.

I dedicate this book to my wife Sushila who is the inspiration in all my work.

E Balagurusamy

# Contents

CHAPTER 1

# Introduction to Numerical Computing

## 1.1 INTRODUCTION

Numerical computations play an indispensable role in solving real life mathematical, physical and engineering problems. They have been in use for centuries even before digital computers appeared on the scene. Great mathematicians like Gauss, Newton, Lagrange, Fourier and many others in the eighteenth and nineteenth centuries developed numerical techniques which are still widely used. The advent of digital computers has, however, enhanced the speed and accuracy of numerical computations.

What is numerical computing? It is important to understand the answer to this fundamental question before we proceed further. *Numerical computing* is an approach for solving complex mathematical problems using only simple arithmetic operations. The approach involves formulation of mathematical models of physical situations that can be solved with arithmetic operations. It requires development, analysis and use of algorithms.

Numerical computations invariably involve a large number of arithmetic calculations and, therefore, require fast and efficient computing devices. The microelectronics revolution and the subsequent development of high power, low cost personal computers have had a profound impact on the application of numerical computing methods to solve scientific problems.

The traditional numerical computing methods usually deal with the following topics:

1. finding roots of equations

2. solving systems of linear algebraic equations
3. interpolation and regression analysis
4. numerical integration
5. numerical differentiation
6. solution of differential equations
7. boundary value problems
8. solution of matrix problems

In this book we will discuss some of the popular methods available in each of these areas.

## 1.2 NUMERIC DATA

Numerical computing may involve two types of data, namely, *discrete data* and *continuous data*. Data that are obtained by counting are called discrete data. Examples of discrete data are the total number of items in a box, or the total number of people participating in a race.

Data that are obtained through measurement are called continuous data. Examples of continuous data are the speed of a vehicle as given by a speedometer, or temperature of a patient as measured by a thermometer.

## 1.3 ANALOG COMPUTING

Analog refers to the principle of solving a problem by ~~using a~~ tool which operates in a way analogous to the problem. For example, the electronic circuits in an analog computer act analogously to the problem to be solved. Analog computing is based on inputs that vary continuously, such as current, voltage or temperature. The earliest computers were analog and functioned on the basis of electrical voltages. Calculations were performed by adding, subtracting, multiplying and dividing voltages. Analog computers are fast, but their accuracy is limited by the precision with which the physical quantities can be read.

Many real life measurable quantities are analog in nature: time, temperature, pressure, and speed, for instance. Analog methods are preferred when these quantities have to be represented in a calculation. An example of application of analog computers is a machine used in a postal department to convert the weight of a package into the cost of postage needed for mailing.

The basic requirement in the application of analog computers is the writing down of differential equations describing the physical system of interest. Given the differential equations, the analog result may be obtained either by direct method, in which equivalent electrical circuits are directly used to simulate the time variations of the dependent variables of the physical system, or the functional method, in which electronic circuits perform the mathematical operations indicated by the terms of the differential equation.

## 1.4 DIGITAL COMPUTING

A digital computer is a computing device that operates on inputs which are discrete in nature. The input data are numbers (or digits) that may represent numerals, letters, or other special symbols. Just as a digital clock directly counts the seconds and minutes in an hour, a digital computer counts discrete data values to compute the results.

Today's digital computers can cope with the analog information, but they have to convert it into digital form. They do this by measuring the value of analog quantity at regular intervals and converting that measurement into a number of electrical pulses corresponding to that measurement. In an analog watch, for example, time and hands on the watch face change continuously; a digital watch, however, converts the passage of time into tiny intervals, marked by the numbers changing on the dial.

Digital computers are more accurate than analog computers. Analog computers may be accurate to within 0.1 per cent of the correct value, whereas digital computers can obtain whatever degree of accuracy is required by choosing the correct number of decimal places. They are designed to read, store, manage, and output specific units like numbers, letters, or punctuation marks. Digital computers are widely used for many different applications and are often called *general purpose computers*.

## 1.5 PROCESS OF NUMERICAL COMPUTING

As stated earlier, numerical computing involves formulation of mathematical models of physical problems that can be solved using basic arithmetic operations. The process of numerical computing can be roughly divided into the following four phases which are illustrated in Fig. 1.1:

1. formulation of a mathematical model
2. construction of an appropriate numerical method
3. implementation of the method to obtain a solution
4. validation of the solution

The formulation of a suitable mathematical model is critical to the solution of the problem. A mathematical model can be broadly defined as a formulation of certain mathematical equation that expresses the essential features of a physical system or process. Models may range from a simple algebraic equation to a complex set of differential equations. Figure 1.2 shows various types of mathematical equations that might result while formulating mathematical models of physical processes.

The formulation of a mathematical model begins with a statement of the problem and the associated factors to be considered. The factors may concern the balance of forces and other laws of conservation in physics.

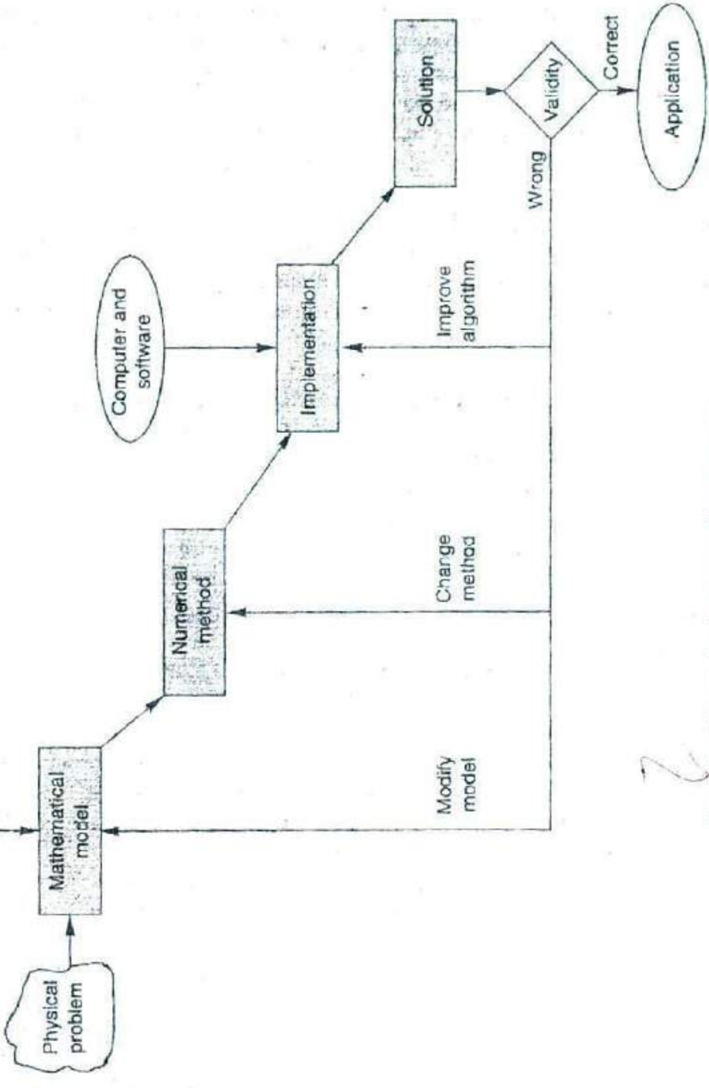


Fig. 1.1 Numerical computing process

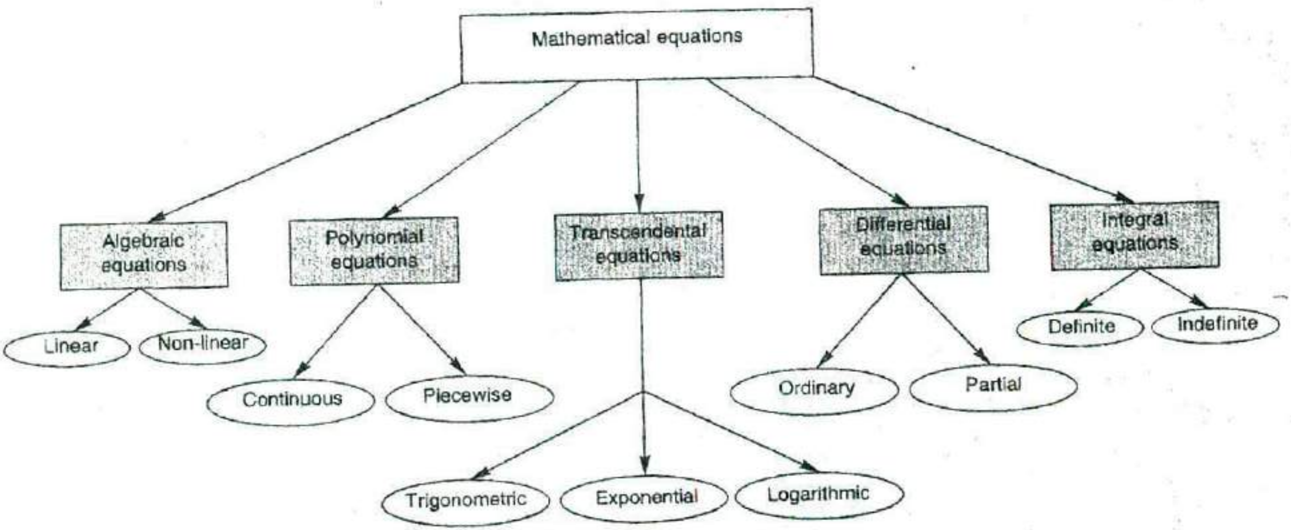


**Fig. 1.2** Different forms of mathematical equations

Real life problems have many uncertainties and unknowns. It might, therefore, be necessary to make certain assumptions for approximating and to include only those features of the problem that are considered critical to the final solution. An over-simplified model may have only limited usefulness. The model may be enhanced later, if necessary. The model refinement may make the solution procedure more difficult. We must always maintain the balance of enhancement of the model and accuracy of the solution required.

Once a mathematical model is available, our first step would be to try to obtain an explicit analytical solution. In most cases, the mathematical models may not be amenable to analytical solutions or they may not be solved efficiently using analytical techniques. In such cases, we have to construct appropriate numerical methods to solve mathematical models. As mentioned earlier, a numerical method is a computational technique which involves only a finite number of basic arithmetic operations.

For a given problem, there might be several alternative numerical methods. We must consider different factors or trade-offs before selecting a particular method—such as type of equation, type of computer available, accuracy, speed of execution, and programming and maintenance efforts required.

Modelling is the process of translating a physical problem into a mathematical problem. The process involves

1. making a number of simplifying assumptions
2. identification of important variables
3. postulation of relationships between the variables

This book is mainly concerned with the solution of mathematical models using numerical techniques.

### Example 1.1

Formulate a mathematical model for predicting the population growth of a city.

*Assumptions:*

Birth and death rates are proportional to population and time interval.

*Parameters:*

$P(t)$ —population at time $t$

$\Delta P$ —increase in population in time interval $\Delta t$

Then,

$$\begin{aligned}\Delta P &= \text{births in } \Delta t - \text{deaths in } \Delta t \\ &= C_1 P(t)\,\Delta t - C_2 P(t)\,\Delta t \\ &= (C_1 - C_2)\,P(t)\,\Delta t\end{aligned}$$

$$\text{Growth rate} = \frac{\Delta P}{\Delta t} = CP(t)$$

Taking the limits $\Delta t \to 0$, we get,

$$\frac{dp}{dt} = C\,P(t)$$

Solution of this differential equation is

$$P(t) = P_0\,e^{ct}$$

where $P_0$ is the population at time $t = 0$.

The population growth depends on the growth constant $C = C_1 - C_2$. The population will be stable if $C_1 = C_2$.

The third phase of the numerical computing process is the implementation of the method selected. This phase is concerned with the following three tasks

1. design of an algorithm
2. writing of a program
3. executing it on a computer to obtain the results

Once we are able to obtain the results, the next step is the validation of the process. Validation means the verification of the results to see that it is within the desired limits of accuracy. If it is not, then we must go back and check each of the following:

1. mathematical model itself
2. numerical method selected
3. computational algorithm used to implement the method

This may mean modification of the model, selection of an alternate numerical method or improving the algorithm (or a combination of them). Once a modification is introduced, the cycle begins again. Figure 1.3 illustrates how the numerical computing cycle moves from the real world to mathematical world and back.

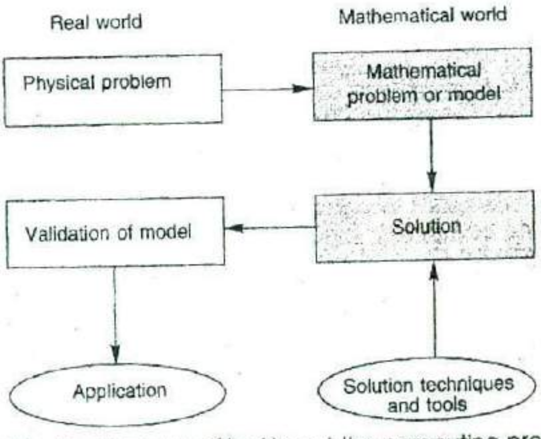


Fig. 1.3 Another way of looking at the computing process

## CHARACTERISTICS OF NUMERICAL COMPUTING

Numerical methods exhibit certain computational characteristics during their implementation. It is important to consider these characteristics while choosing a particular method for implementation. The characteristics that are critical to the success of implementation are: accuracy, rate of convergence, numerical stability, and efficiency.

### Accuracy

Every method of numerical computing introduces errors. They may be either due to using an approximation in place of an exact mathematical procedure (known as *truncation errors*) or due to inexact representation and manipulation of numbers in the computer (known as *roundoff errors*). These errors affect the accuracy of the results. The results we obtain must be sufficiently accurate to serve the purpose for which the mathematical model was built. Choice of a method is, therefore, very much dependent on the particular problem. The general nature of these errors will be discussed in detail in Chapter 4.

### Rate of Convergence

Many numerical methods are based on the idea of an *iterative process*. This process involves generation of a sequence of approximations with the hope that the process will converge to the required solution. Certain methods converge faster than others. Some methods may not converge at all. It is, therefore, important to test for convergence before a method is used. Rapid convergence takes less execution time on the computer. There are several techniques for accelerating the rate of convergence of certain methods. The concepts of convergence and divergence are discussed in Chapter 4. They are also discussed in various places where specific methods are analysed for convergence.

### Numerical Stability

Another problem introduced by some numerical computing methods is that of *numerical instability*. Errors introduced into a computation, from whatever source, propagate in different ways. In some cases, these errors tend to grow exponentially, with disastrous computational results. A computing process that exhibits such exponential error growth is said to be numerically unstable. We must choose methods that are not only fast but also stable.

Numerical instability may also arise due to ill-conditioned problems. There are many problems which are inherently sensitive to round off errors and other uncertainties. Thus, we must distinguish between sensitivity of methods and sensitivity inherent in problems.

When the problem is ill-conditioned, there is nothing we can do to make a method to become numerically stable.

### Efficiency

One more consideration in choosirg a numerical method for solution of a mathematical model is *efficiency*. It means the amount of effort required by both human and computer to implement the method. A method that requires less of computing time and less of programming effort and yet achieves the desired accuracy is always preferred.

## 1.7 COMPUTATIONAL ENVIRONMENT

The last phase of the numerical computing process, namely the implementation phase, requires resources such as computer hardware, operating system and other systems software, language compilers, actual application programs and other software tools to manipulate data and provide output in a desired form.

The computer hardware may range from a small personal computer to a large super computer depending on the nature and size of the problem. A program may not always produce the same results on two different types of computers due to difference in their round off errors.

Appropriate operating systems and compilers play an important role in developing portable programs. UNIX and MS-DOS have become popular operating systems for scientific computing. FORTRAN language has dominated the scientific computing field for the last four decades and it is expected to continue its predominant role for some more years. It has been continuously modified and extended to support the ever changing requirements of software engineering. The likely strong competitor for FORTRAN in the near future will be C and C++ languages which contain some unique features and powerful control structures. Portability is another strong point of these languages.

## 1.8 NEW TRENDS IN NUMERICAL COMPUTING

In recent years, the increasing power of computer hardware has affected the approach of numerical computing in several ways. It has forced scientists and engineers to search for algorithms that are computationally fast and efficient. An important new trend is the construction of algorithms to take advantage of specialised computer hardware such as *vector computers* and *parallel computers*. Another trend is the use of sophisticated interactive graphics, in which the user can view the results graphically and advise the computer, graphically, on how to proceed further.

One important development which is likely to have an increasing impact on scientific computing is *symbolic computation*. Symbolic computation systems would enable us to add, multiply and divide polynomials or rational expressions the same way we would do using pencil and paper. They can also solve certain mathematical problems without rounding off errors. Symbolic computation is expected to play an increasing role in scientific computation.

Object-oriented numerical computing is gaining importance due to the popularity of languages like C++ and Java. They incorporate concepts such as encapsulation, inheritance, polymorphism and operator overloading. They support the idea that program units should interact with one another only through clearly defined interfaces. They also enable the extension (or reuse) of the existing code without modifying it.

## 1.9 MATHEMATICAL BACKGROUND

This book assumes that the readers have some mathematical background. They require basic knowledge of algebra, functions, matrices, and integral and differential calculus.

## 1.10 SUMMARY

In this chapter, we have introduced the concept of numerical computing and discussed the steps involved in solving a physical problem using numerical methods. We also discussed the characteristics of numerical computing and computing resources required for implementing a numerical method.

### Key Terms

*Accuracy*
*Algorithm*
*Analog computer*
*C*
*C++*
*Continuous data*
*Digital computer*
*Discrete data*
*Efficiency*
*FORTRAN*
*General purpose computers*
*Ill-conditioned problems*
*Iterative process*
*Mathematical model*
*Numerical computing*
*Numerical method*
*Numerical stability*
*Parallel computers*
*Rate of convergence*
*Round off error*
*Symbolic computation*
*Truncation error*
*Validation*
*Vector computer*.

### REVIEW QUESTIONS

1. What is Numerical Computing?
2. Distinguish between analog computing and digital computing.
3. Describe, with the help of a block diagram, the process of numerical computing.

4. Newton's second law of motion states that the time rate of change of momentum of a body is equal to the resultant force acting on it. Using this law, formulate a mathematical model to determine the terminal velocity of a free falling body near the earth's surface.
5. The Newton's law of cooling states that the rate of heat from a liquid is proportional to the difference in temperatures between the liquid and the surroundings. Formulate a mathematical model to govern this law.
6. When a boat moves through water, the retarding force is proportional to the square of the velocity. Formulate a differential equation in terms of velocity given the mass $m$ and the drag coefficient $k$.
7. State the four characteristics of numerical computing.
8. What is accuracy? How is it affected during the process of numerical computing?
9. What is convergence? How is it important in numerical computing?
10. What do you mean by numerical instability?
11. Distinguish between sensitivity of methods and sensitivity of problems.
12. Describe resources required for implementing a numerical computing process.

CHAPTER 2

# Introduction to Computers and Computing Concepts

## 2.1 INTRODUCTION

In Chapter 1, we discussed that numerical computing requires two important tools, namely, mathematical methods and computers. Most numerical methods cannot be solved without the help of computers. Therefore, a background knowledge of computers and computing concepts will enhance the understanding of implementation of numerical computing solutions. This chapter provides some basic information on computing en ironment and problem solving approach using computers.

The spate of innovations and inventions in computer technology during the last two decades has led to the development of a variety of personal computers. They are so versatile that they have become indispensable to engineers, scientists, business executives, managers, administrators, accountants, teachers and students. They have strengthened humankind's powers in numerical computations and information processing.

Modern computers possess certain characteristics and abilities peculiar to them. They can

1. perform complex and repetitive calculations rapidly and accurately
2. store large amounts of data and information for subsequent manipulations
3. hold a program of a model which can be explored in many different ways
4. make decisions
5. provide information to the user

6. automatically correct or modify certain parameters of a system under control
7. draw and print graphs
8. converse with users interactively

Engineers and scientists make use of the high-speed computing capability of computers to solve complex mathematical models and design problems. Many calculations that were previously beyond contemplation have now become possible. But for computers, many of the technological achievements, such as landing on the moon, would not have been possible.

Computers have helped automation of many industrial and business systems. They are used extensively in manufacturing and processing industries, power distribution systems, airline reservation systems, transportation systems, banking systems, and so on. *Computer-aided design* (CAD) and *computer-aided manufacture* (CAM) are among the most popular industrial applications today.

*Modelling* and *simulation* is another area where computers are increasingly used. This has greatly accelerated research in such areas as physical and social sciences, medicine, astronomy and meteorology.

Business and commercial organisations need to store and maintain voluminous records and use them for various purposes such as inventory control, sales analysis, payroll accounting, resources scheduling and generation of management reports. Computers can store and maintain files and can sort, merge or update them as and when necessary.

The ability of computers to store large amounts of data has led to their application in libraries, documentation centres, employment exchanges, police departments, hospitals and other similar establishments. Computers are used in international games such as the Olympics to keep track of events and provide timely and reliable information and documentation to all concerned.

Since computers can bank a variety of information and converse with the users, they are being used as resources in teaching and learning at all levels of education and training. This process is known as *computer-assisted learning* (CAL). Here, learners can communicate directly with a computer in a conversational mode. Using this mode, a learner can learn a topic in his own time and pace.

Computers are also used to manage the learning processes. This is called *computer-managed learning* (CML). Computers can store students' responses, evaluate their performance and then direct them to the next appropriate learning unit.

The areas of computer applications are too numerous to mention. Computers have become an integral part of our everyday life. They continue to grow and open new horizons of discovery and application such as the electronic office, electronic commerce, and the home computer centre.

The microelectronics revolution has placed enormous computational power within the reach of every scientist and engineer. However, it must be remembered that computers are machines created and managed by humans. A computer has no brain of its own. Anything it does is the result of human instructions. It is an obedient slave which carries out the master's orders as long as it can understand them, no matter whether they are right or wrong. In short, computers lack common sense. These instructions constitute the program or software.

## 2.2 EVOLUTION OF NUMERICAL COMPUTING AND COMPUTERS

The use of computing techniques is over 5000 years old. The Babylonians, Chinese, and Egyptians used numerical methods for the survey of lands and the collection of taxes as early as 3000 BC. Computing history starts with the development of a device called the *abacus* by the Chinese around this period. This was used for the systematic calculation of arithmetic operations. Since then the number system has undergone various changes and has been used in different forms in computing. The most significant development in computing was the formulation of the decimal number system in India around 800 AD. Another significant development was the invention of *logarithm* by John Napier in 1614, which made computing simple.

The modern age of mathematics emerged during the 17th century when Johannes Kepler and Galileo Galilee deduced the laws for planetary motion and Sir Isaac Newton formulated the law of gravity. The subsequent developments in mathematics and other sciences increased the need for new computing techniques and devices.

The principle of logarithm was later applied to a calculating device known as the *slide rule*, which was extensively used till recently. The first accounting machine was built in France by Blaise Pascal in 1642. Then came the Leibnitz calculator in 1671 designed by Gottfried Wilhelm von Leibnitz. These machines progressed in technology and variety and became the standard calculating machines of the business community. During the beginning of the 19th century, Joseph Marie Jacquard invented an automated loom operated by a mechanism controlled by punched cards.

The origin of the modern computer can be traced back to 1834 AD, when an English mathematician, Charles Babbage, designed an analytical engine. This is considered to be the first programmable digital mechanical computer. However, this kind of machine was not built until 1944, when Mark I, an electromechanical automatic computer, was developed by IBM. Subsequently, a series of technological improvements and innovations took place and the design of computers underwent continuous and dramatic changes. Some of the important developments since the slide rule are given in Table 2.1.

**Table 2.1** Some developments in computing technology

| *Year* | *Device* |
|---|---|
| 1622 | Slide rule |
| 1642 | Pascal calculator, an accounting machine by Blaise Pascal |
| 1671 | Leibnitz calculator |
| 1801 | Punched card loom by Jacquard |
| 1822 | Difference engine by Charles Babbage |
| 1834 | Analytical engine by Charles Babbage |
| 1890 | Punched card machine by Herman Hollerith |
| 1930 | Differential analyser by Vannevar Bush |
| 1936 | Paper on computational numbers by Alan Turing<br>Link between symbolic logic and electric circuit by Claude Shanon |
| 1937 | Binary adder built by George Stibitz |
| 1941 | First general-purpose computer designed by Konrad Zuse |
| 1943 | Colossus machine built to crack German secret codes, by the British |
| 1944 | First automatic computer, MARK I, designed by Howard Aiken |
| 1945 | Critical elements of a computer system outlined by John Von Neumann |
| 1946 | First electronic digital computer, ENIAC, put to operation by Presper Eckert and John Mauchly |
| 1947 | Transistor invented by John Bardeen, William Shockley and Walter Brattain |
| 1951 | First business computer, UNIVAC, became operational |
| 1956 | Second generation computer (using transistors) introduced by Bell Laboratory |
| 1959 | Integrated circuits (ICs) demonstrated by Clair Kilby |
| 1964 | First third generation computer using ICs developed |
| 1965 | First commercial minicomputer, PDP-8, introduced by Digital Equipment Corporation |
| 1971 | Intel 4004 microprocessor designed by Ted Hoff |
| 1974 | First fourth generation computer (using microprocessors) built by Ed Roberts |
| 1975 | First personal computer software created by Bill Gates and Paul Allen |
| 1977 | Apple introduced its famous personal computer |
| 1981 | IBM PC introduced in the market |
| 1982 | Cray supercomputer marketed by Cray Research Company |
| 1989 | Optical computer demonstrated |

## Modern Computers

The era of modern computers began in 1951 when the UNIVAC (Universal Automatic Computer) became operational at the Bureau of Census in USA. Since then, computers started appearing in quick succession, each claiming an improvement over the other. They represented improvements in speed, memory (storage) systems, input and output devices and programming techniques. They also showed a continuous reduction in physical size and cost. The developments in computers are closely associated with the developments in material technology, particularly the semiconductor technology.

Computers developed after ENIAC have been classified into the following four generations:

| | |
|---|---|
| First generation | 1946 – 1955 |
| Second generation | 1956 – 1965 |
| Third generation | 1966 – 1975 |
| Fourth generation | 1976 – present |

You may notice that from 1946, each decade has contributed one generation of computers.

In the *first generation* computers vacuum tubes were used. Magnetic tape drives and magnetic core memories were developed during this period. The first generation computers possessed the following drawbacks as compared to the later models:

1. large in size
2. slow operating speeds
3. restricted computing capacity
4. limited programming capabilities
5. short life span
6. complex maintenance schedules

The *second generation* computers were marked by the use of a solid-state device, called the transistor, in the place of vacuum tubes. These machines were much faster and more reliable than their earlier counterparts. Further, they occupied less space, required less power, and produced much less heat.

Research in the field of electronics led to the innovation of the integrated circuits, now popularly known as IC chips. The use of IC chips in the place of transistors gave birth to the *third generation* computers. They were still more compact, faster and less expensive than the previous generation.

Along with the third generation computers, newer and faster equipments were introduced for handling storage and input-output.

Continued efforts towards miniaturisation led to the development of large-scale integration (LSI) technology. Intel Corporation introduced LSI chips called microprocessors for building computers. The latest child of the computer family that uses VLSI chips has been named the *fourth generation* computer. The fourth generation computers are marked with an increased user-computer interaction and speed. Table 2.2 gives an idea of the main features of each generation.

## Fifth Generation Computers

Japan and many other countries are working on systems that are known as *knowledge-based* or *expert systems* which will considerably improve the man-machine interaction. Such systems would integrate the advancements in both hardware and software technologies and would facilitate computer-aided problem-solving with the help of organised information in many specialised areas.

**Table 2.2** Computer generations

| *Features* | *Generation* | | | |
|---|---|---|---|---|
| | *First* | *Second* | *Third* | *Fourth* |
| Main component | Vacuum tube | Transistor | Integrated circuit (IC Chips). | LSI and VLSI circuit |
| Internal storage (Memory) | Electrostatic tubes | Magnetic core<br>Magnetic drum | Magnetic core | Semiconductor memory |
| External storage (Auxiliary memory) | Paper tape<br>Punched card<br>Magnetic tape | Magnetic disk<br>Magnetic drum<br>Magnetic tape<br>Paper tape<br>Punched card | Magnetic disk<br>Magnetic tape<br>Magnetic drum<br>Punched card<br>Paper tape | Magnetic disk<br>Magnetic tape<br>Magnetic drum<br>Floppy disk<br>CD rom |
| Speed of operation (Additions/second) | 40 to 300 thousands | 3,000 to 30,000 thousands | 30,000 to 3,00,000 thousands | 3,00,000 to 30,00,000 thousands |

This generation of computers is called the *fifth generation computers*. Although knowledge-based systems are expensive and time-consuming to build, they are likely to become more popular in the coming years.

## 2.3 TYPES OF COMPUTERS

Computers may be classified based on operating principles, size and capability, and applications.

### Principles of Operation

Based on the operating principles, computers can be classified into any one of the following types: digital computers, analog computers, and hybrid computers.

*Digital computers* operate essentially by counting. All quantities are expressed as discrete digits or numbers. Digital computers are useful for evaluating arithmetic expressions and for manipulations of data (such as preparation of bills, ledgers, solution of simultaneous equations, etc.).

*Analog computers* operate by measuring rather than by counting. The name, which is derived from the Greek word *analog*, denotes that the computer functions by establishing similarities between two quantities that are usually expressed as voltages or currents. Analog computers are powerful tools to solve differential equations. Computers which combine features of both analog and digital types are called *hybrid computers*.

A majority of the computers used today are digital. As their name suggests, digital computers were originally designed to perform certain numerical calculations. They gradually replaced almost all mechanical calculating devices. Later, the concept of stored programs enabled them to store data and instructions and perform certain sequences and combinations of arithmetic operations automatically. This has led to the use of digital computers in a variety of applications.

## Applications

Modern computers, depending upon their applications, are classified as special purpose computers or general purpose computers.

*Special purpose* computers are tailor-made to cater solely to the requirements of a particular task or application. They incorporate the instructions needed into the design of internal storage so that they can perform the given task on a simple command. They, therefore, do not possess unnecessary options and cost less.

On the other hand, *general purpose* computers are designed to meet the needs of many different applications. In a general purpose computer, the instructions needed to perform a particular task are not wired permanently into the internal memory. When one job is over, instructions for another job can be loaded into the internal memory for processing. Thus, a general-purpose machine can be used to prepare pay-bills, manage inventories, print sales reports, and so on.

## Size and Capability

Computers are also available in different sizes and with different capabilities. Broadly, they may be categorised as microcomputers, minicomputers, mainframes and supercomputers. The selection of a particular system primarily depends on the volume of data to be handled and the speed of the processor.

***Microcomputers*** A *microcomputer* is the smallest general-purpose processing system. Functionally, it is similar to any other large system. Microcomputers are self-contained units and are usually designed for use by one person at a time. Since microcomputers can be easily linked to large computers, they form a very important segment of the integrated information systems.

***Minicomputers*** A *minicomputer* is a medium-sized computer that is more costly and powerful than a microcomputer. An important distinction between a microcomputer and a minicomputer is that the latter is usually designed to serve multiple users simultaneously. A system that supports multiple users is called multiterminal, time-sharing system. Minicomputers are the popular computing systems amo ıg research and business organisations today.

***Mainframe computers*** Computers with large storage capacities and very high speed of processing (compared to micro or minicomputers) are known as *mainframe computers*. They support a large number of terminals for use by a variety of users simultaneously. They are also used as the central host computer in distributed data processing systems.

***Supercomputers*** *Supercomputers* have extremely large storage capacities and computing speeds that are many times faster than other computers. While the speed of traditional computers is measured in terms of millions of instructions per second (mips), a supercomputer is rated in tens of millions of operations per second (mops) (an operation is made up of numerous instructions). Typically, the supercomputer is used for large-scale numerical problems in scientific and engineering disciplines. These include applications in electronics, petroleum engineering, weather forecasting, structural analysis, chemistry, medicine and physics.

***Personal computers*** *Personal computers* are nothing but micro- computers that are specially designed for personal use of individuals. The name "personal computer" was coined by IBM when it marketed its first microcomputer in 1981. Since then, many companies have produced IBM compatible PCs. During the last fifteen years, the processor chips used in IBM compatible PCs have undergone dramatic improvements in their performance characteristics. Table 2.3 shows the characteristics of various PC processor chips. Note that today's PCs are far more powerful than the mainframes of just a few years ago.

**Table 2.3** Characteristics of microprocessor chips

| | *8088* | *286* | | | | *Pentium* | *Pentium* |
|---|---|---|---|---|---|---|---|
| | *PC XT* | *PC AT* | *386* | *486* | *Pentium* | *Pro* | *II* |
| Clock speed (megahertz) | 4.77 | 6–12 | 16–33 | 16–50 | 66–200 | 120–200 | 200+ |
| Data path (bits) | 8 | 16 | 32<br>16 (SX) | 32 | 64 | 64 | 64 |
| Computation size (bits) | 16 | 16 | 32 | 32 | 32 | 32 | 32 |
| Memory-size (bytes) | 640K | 2 megs | 4–16 megs | 4–64 megs | 4–64 megs | 16–64 megs | 16–64 megs |
| Floating point | Coprocessor | Coprocessor | Coprocessor | On chip | On chip | On chip | On chip |
| Speed (MIPS) | 0.33 | 1.2 | 2.5–6 | 20–40 | 112 | 250 | 500 |
| Number of transistors per chip | 29,000 | 130,000 | 275,000 | 1.2 million | 3.3 million | 5.2 million | 10+ million |

***Workstations*** There is a class of computers, known as *workstations*, which lie in between minicomputers and microcomputers in terms of processing power. A workstation looks like a personal computer but is specially designed for engineering and graphics applications.

***Parallel computers*** *Parallel computer* is a relatively new type of computer that uses a large number of processors. The processors perform different tasks independently and simultaneously, thus, improving the speed of execution of complex programs dramatically. Parallel computers match the speed of supercomputers at a fraction of the cost.

## 2.4 COMPUTING CONCEPTS

A computer, small or big, is basically a device used for processing of data (numbers) and text (words). It performs essentially the following three operations in a sequence:

1. receives data (and instructions)
2. processes data (as per the instructions)
3. outputs result (information)

This cycle of operation of a computer is known as the *input–process–output* cycle and is shown in Fig. 2.1.

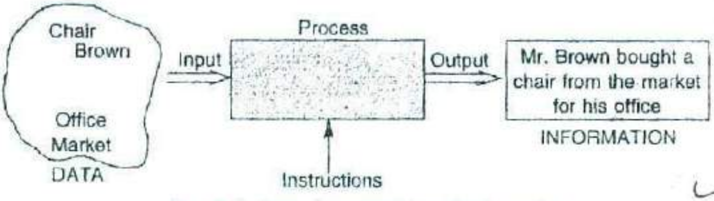


Fig. 2.1 Input-process-output cycle

Raw facts, known as *data,* are provided to the computer in bits and pieces. They are encoded in such a way that the computer can understand them. The computer then processes the data with the help of certain instructions provided to it, and produces a meaningful and desired output known as *information*. For example, if the data consists of two numbers, say, 10 and 15 and the instruction is to add them and print out the result, then the output information would be the sum of the two numbers, i.e. 25. A set of instructions designed to perform a particular sequence of functions is called a *computer program*.

Processing is nothing but manipulation of data in accordance with certain procedures to suit the need of the user (or application). The same basic data can provide several kinds of information depending upon the type of instructions.

Input is usually through a keyboard (like a typewriter) and output may be obtained either on a *display screen* or on a *printer*. While the printer produces typed copy on paper (usually known as *hard copy*), the screen display (*soft copy*) allows the user to verify the output before it is printed.

A computer often includes an external storage system to store (and retrieve) data and programme. The popular storage medium is a floppy disk. Other media, such as hard disks, magnetic tapes and CD ROMs are also used. All these physical components are known as *hardware*.

## 2.[illegible] COMPUTER ORGANISATION

Although computers differ widely in their details, all of them follow a basic organisational structure as shown in Fig. 2.2. In order to carry out the three basic operations, namely, input, process and output, a computer includes the following hardware components: input devices, processing units, output devices and external storage devices.

KGTG

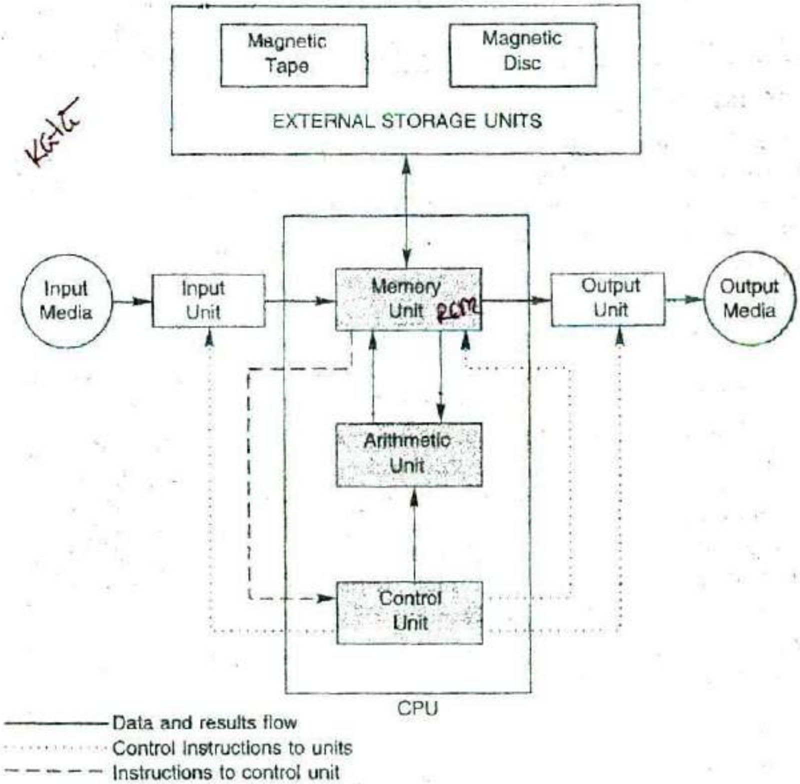


**Fig. 2.2** Structure of a computer

### Input Devices

An input device presents data to the processing unit in machine-readable form. Although the keyboard is a common input device for a small computer, a system may also support one or more of the input devices given in Table 2.4.

**Table 2.4** Input devices

| *S.No.* | *Device* | *Medium of data storage* | *Remarks* |
|---|---|---|---|
| 1. | Optical character reader (OCR) | Special paper document | Only input |
| 2. | Magnetic ink character recogniser (MICR) | Special paper document | Only input |
| 3. | Mark sense reader | Special paper or card | Only input |
| 4. | Graphics tablet | Document | Only input |
| 5. | Mouse | Document | Only input |
| 6. | Floppy drive | Floppy disk | Input, output, storage |
| 7. | Hard disk (Winchester) drive | Magnetic disk | Input, output, storage |
| 8. | Tape drive | Magnetic tape | Input, output, storage |
| 9. | CD ROM drive | CD ROM | Input, storage |

## Processing Units

Processing units receive data and instructions, store them temporarily and then process the data as per the instructions. The processing units include: memory unit, arithmetic logic unit, and control unit. All three units together are known as the *central processing unit* (CPU).

***Memory unit*** The memory unit holds (stores) all data, instructions and results temporarily. The memory consists of hundreds of thousands of cells called 'storage locations', each capable of storing one word of information. The memory unit is called by different names, such as storage, internal storage, primary storage, main memory or simply memory.

***Arithmetic logic unit*** This unit is used to perform all the arithmetic and logic operations, such as addition, multiplication, comparison, etc. For example, consider the addition of two numbers $A$ and $B$. The control unit will select the number $A$ from its location in the memory and load it into the arithmetic logic unit. Then it will select the number $B$ and add it to $A$ in the arithmetic unit. The result will then be stored in the memory or retained in the arithmetic unit for further calculations.

***Control unit*** This unit coordinates the activities of all the other units in the system. Its main functions are:

1. to control the transfer of data and information between various units
2. to initiate appropriate actions by the arithmetic unit

The program provides the basic control instructions. Conceptually, the control unit fetches instructions from the memory, decodes them, and directs various units to perform the specified tasks.

## Output Devices

Output devices receive information from the CPU and present it to the user in the desired form. Although a printer is the most commonly used output device, devices such as plotters are also becoming pcpular. Some common output devices are given in Table 2.5.

**Table 2.5** Output devices

| *Device* | *Medium of presentation* | *Remarks* |
|---|---|---|
| Printer | Paper | Only output |
| Plotter | Paper | Only output |
| Visual display unit (VDU) | Display screen | Only output |
| Floppy drive | Floppy disk | Input, output, storage |
| Disk drive | Magnetic disk | Input, output, storage |
| Tape drive | Magnetic disk | Input, output, storage |

## External Storage Devices

The purpose of external storage is to retain data and programs for future use. For example, a program may be required at regular intervals. If such information is stored in an external storage media, then one can retrieve it as and when necessary, thus avoiding the need to type it again. Any number of files containing information can be stored on external media. Since they are permanent (they are not erased when the equipment is turned off), one can store long files on external media, and later on work on them in sections, keeping all the sections in storage except the one currently in use.

The popular external storage media used with micro and mini computers are floppy disks, hard disks and CD ROMs.

***Floppy disks*** The most common storage medium used on small computers today is a *floppy disk*. It is a flexible plastic disk coated with magnetic material and looks like a phonograph record. Information can be recorded or read by inserting it into a disk drive connected to the computer. The disks are permanently encased in stiff paper jackets for protection and easy handling. An opening is provided in the jacket to facilitate reading and writing of information (Fig. 2.3).

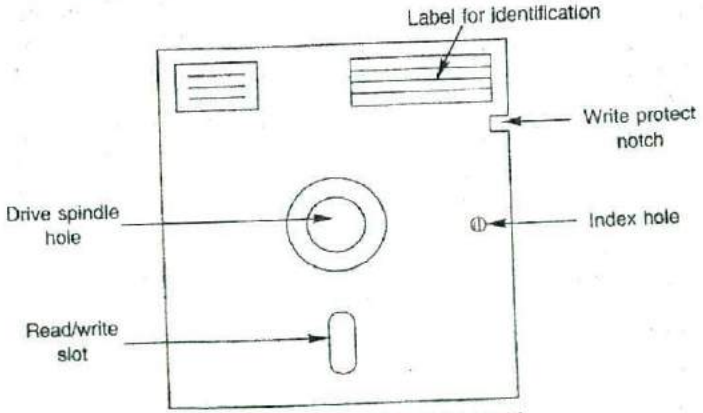


**Fig. 2.3** Floppy disk (5.25 inch)

Floppy disks are available in two standard sizes—5.25 inch and 3.5 inch. The 3.5 inch floppy disk, which was introduced later, can store more information than the previous one.

***Hard disks*** Another magnetic media suitable for storing large volumes of information is the *hard disk*, popularly known as the *Winchester disk*. A hard disk pack consists of two or more magnetic plates fixed to a spindle, one below the other, with a set of read/write heads. The disk pack is permanently sealed inside a casing to protect it from dust and other contaminations, thus increasing its operational reliability and data integrity.

Winchester disks possess a number of advantages compared to floppy disks:

1. They can hold much larger volumes of information than floppies.
2. They are very fast in reading and writing.
3. Hard disks are not susceptible to dust and static electricity.

Winchester disks are available in different sizes and capacities. Standard sizes are 5.25 inch, 8 inch, 10.5 inch and 14 inch. Storage capacities of 260, 540, 680, 1000, 1200, 2000 megabytes are typical on a personal computer.

***CD ROMs*** Compact disk read-only memory (CD ROM) disks are used to distribute large volumes of data and text. Computer programs and user manuals are often distributed on CD ROMs.

## 2.6 DRIVING THE COMPUTER: THE SOFTWARE

Computers need clear-cut instructions to tell them *what to do, how to do*, and *when to do*. A set of instructions to carry out these functions is called a *program*. A group of such programs that are put into a computer to operate and control its activities is called the *software*. These programs must reside in the internal storage (memory) to execute their instructions. For example, if we want to delete some data stored in memory, the system uses one set of program instructions. Similarly, if we want to sort a list of names, it uses another set of instructions designed to perform this task.

Software is an essential requirement of computer systems. Just as a car cannot run without fuel, a computer cannot work without software. There are four major kinds of software that are implemented as shown in Fig. 2.4: operating system, utility programs, language processors and application programs.

Software is intangible but resides on or is stored in something tangible, such as floppy disks and magnetic tapes.

### Operating System

The software that manages the resources of a computer system and schedules its operation is called the *operating system*. The operating

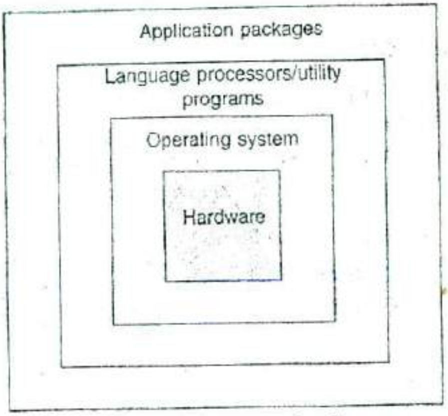


**Fig. 2.4** Layers of software

system acts as an interface between the hardware and the user programs and facilitates the execution of the programs (Fig. 2.4). The principal functions of operating system include:

1. to control and coordinate peripheral devices such as printers, display screen and disk drives
2. to monitor the use of the machine's resources
3. to help the application programs execute its instructions
4. to help the user develop programs
5. to deal with any faults that may occur in the computer and inform the operator

The operating system is usually available with hardware manufacturers and is rarely developed in-house owing to its technical complexity. Small computers are built from a wide variety of microprocessor chips and use different operating systems. Hence, an operating system that runs on one computer may not run on the other. The popular operating systems include, among others, MS DOS and UNIX.

## Utility Programs

There are many tasks common to a variety of applications. Examples of such tasks are:

1. sorting a list in a desired sequence
2. merging of two programs
3. copying a program from one place to another
4. report writing

One need not write programs for these tasks. They are standard, and normally handled by *utility programs*.

Like operating systems, utility programs are pre-written by the manufacturers and supplied with the hardware. They may also be obtained from standard software vendors. A good range of utility programs can make life much easier for the user.

## Language Processors

Computers can understand instructions only when they are written in their own language called the *machine language*. Therefore, a program written in any other language should be translated into machine language. Special programs called *language processors* are available to do this job.

These special programs accept the user programs and check each statement and, if it is grammatically correct, produce a corresponding set of machine code instructions. Language processors are also known as *translators*.

There are two forms of translators: compilers and interpreters.

A *compiler* checks the entire user-written program (known as the *source program*) and, if error-free, produces a complete program in machine language (known as *object program*). The source program is retained for possible modifications and corrections and the object program is loaded into the computer for execution.

An *interpreter* does a similar job but in a different style. The interpreter (as the name implies) translates one statement at a time and, if error-free, executes the instruction. This continues till the last statement. Thus an interpreter translates and executes the first instruction before it goes to the second, while a compiler translates the entire program before execution.

The major differences between a compiler and an interpreter are:

1. Error correction (called *debugging*) is much simpler in the case of the interpreter because it is done in stages. The compiler produces an error list for the entire program at the end.
2. Interpreters take more time for the execution of a program compared to compilers because a statement has to be translated every time it is read.

Compilers and interpreters are usually written and supplied by the hardware vendors. Since a compiler (or an interpreter) can translate only a particular language for which it is designed, one will need to use a separate translator for each language.

## Application Programs

While an operating system makes the hardware run properly, *application programs* make the hardware do useful work. Application programs are specially prepared to do certain specific tasks. They can be classified into two categories: standard applications, and unique applications.

Some applications are common for many organisations. Ready-to-use software packages for such applications are available from hardware and/or software vendors. Standard packages include, among others, Sales Ledger, Purchase Ledger, Statistical Analysis, Pay Roll, PERT/CPM, Production Planning and Control, Inventory Management, and Linear Programming.

In some situations one may have to develop one's own programs to suit one's unique requirements. Once developed, they come into the category of unique application packages.

## 2.7 PROGRAMMING LANGUAGES

The functioning of a computer is controlled by a set of instructions (called a *computer program*). These instructions are written to tell the computer:

1. what operation to perform
2. where to locate data
3. how to present results
4. when to make certain decisions

The communication between two parties, whether they are machines or human beings, always needs a common language or terminology. The language used in the communication of computer instructions is known as the programming language. The computer has its own language and any communication with the computer must be in its language or translated into this language.

Three levels of programming languages are available. They are:

1. machine languages (low level languages)
2. assembly (or symbolic) languages
3. procedure-oriented languages (high level languages)

### Machine Language

Computers are made of two-state electronic components which can understand only pulse and no-pulse (or '1' and '0') conditions. Therefore, all instructions and data should be written using *binary codes* 1 and 0. The binary code is called the *machine code* or *machine language*.

Computers do not understand English, Hindi or Tamil. They respond only to machine language. Added to this, computers are not identical in design. Therefore, each computer has its own machine language. (However, the script 1 and 0, is the same for all computers). This poses two problems for the user.

First, it is difficult to understand and remember the various combinations of 1's and 0's representing numerous data and instructions. Also, writing error-free instructions is a slow process.

Secondly, since every machine has its own machine language, the user cannot communicate with other computers (if he does not know its language). Imagine a Tamilian making his first trip to Delhi. He would face enormous obstacles as the language barrier would prevent him from communicating.

### Assembly Language

An *assembly language* uses mnemonic codes rather than numeric codes (as used in machine language). For example, ADD or A is used as a symbolic operation code to represent addition and SUB or S is used for subtraction. Memory locations containing data are given names such as TOTAL, MARKS, TIME, MONTH, etc.

As the computer understands only machine code instructions, a program written in assembly language must be translated into machine language before the program is executed. This translation is done by a computer program referred to as an *assembler*.

The assembly language is again a machine-oriented language and hence, the program has to be different for different machines. The programmer should remember machine characteristics when he prepares a program. Writing a program in assembly language is still a slow and tedious task.

## Procedure-Oriented Language (POL)

These languages consist of a set of words and symbols and one can write programs using these in conjunction with certain rules. These languages are oriented toward the problem to be solved or procedures for solution rather than mere computer instructions. These are more user-centered than the machine-centered languages. They are better known as *high-level languages*.

The most important characteristic of a high-level language is that it is machine-independent and a program written in a high-level language can be run on computers of different makes with little or no modification. The programmer need not know the characteristics of that machine. However, such programs need to be translated into equivalent machine-code instructions before actual implementation.

A program written in a high-level language is known as the *source program* and can be run on different machines using different translators. The translated program is called the *object program*. The major disadvantage of high-level languages is that they take extra time for conversion and thus, are less efficient compared to the machine-code languages. Figure 2.5 shows the system of implementing the three levels of languages.

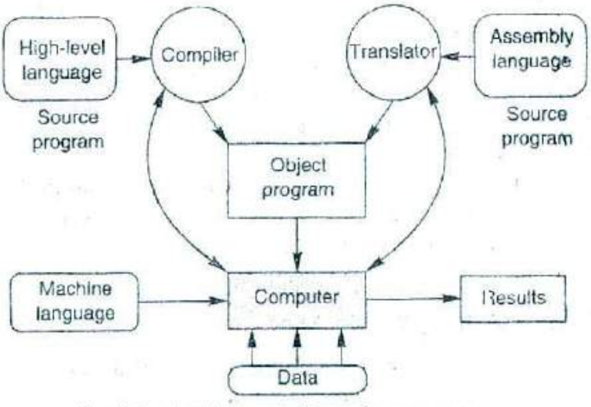


Fig. 2.5 Implementation of a program

## Common High-level Languages

Many high-level languages have been developed during the last three decades. The most common high-level languages are FORTRAN, BASIC, COBOL, C, PL/1, C++ and Java. Although they are less efficient than the machine or assembly languages, they relieve the programmers of the tedious task of remembering numeric codes for storage locations, operations, etc. In addition, these languages are easier to learn and use.

The choice of a language depends upon many factors such as the knowledge of the programmer, the computer, the problem to be solved, etc. The languages that are used more popularly are given in Table 2.6.

**Table 2.6** Summary of common high-level languages

| *Year* | *Language* | *Name derived from* | *Developed by* | *Application* |
|---|---|---|---|---|
| 1957 | FORTRAN | FORmula TRANslation | IBM | Science, engineering |
| 1958 | ALGOL | ALGOrithmic Language | International group | Science, engineering |
| 1959 | LISP | LISt Processing | MIT, USA | Artificial engineering |
| 1960 | APL | A Programming Language | IBM | Science, engineering |
| 1961 | COBOL | COmmon Business Oriented Language | Defence Dept., USA | Business |
| 1964 | BASIC | Beginner's All purpose Symbolic Instruction Code | Dartmouth College, USA | Engineering, science, business, education |
| 1965 | PL/1 | Programming Language 1 | IBM | General |
| 1970 | Pascal | Blaise Pascal | Federal Institute of Technology, Switzerland | General |
| 1972 | PROLOG | PROgramming in LOGic | University of Marseille | Artificial intelligence |
| 1973 | C | Earlier language called B | Bell Laboratory | General |
| 1975 | Ada | Augusta Ada Byron | U.S. Defence Dept. | General |
| 1983 | C++ | Language C | Bell Laboratory | General, object-oriented programming |
| 1991 | Java | None | Sun Microsystems | General, internet, object-oriented programming |

## 2.8 INTERACTIVE COMPUTING

A major breakthrough in programming took place in the early 1960s when interactive languages like BASIC were developed. With an interactive language, we can converse (interact) with a computer. Most of the modern languages including FORTRAN have incorporated interactive features. With the help of an interactive language, we may engage in a conversation with our computer like this:

```
I am computing sum of two values
Please input value of X
255.75
Please input value of Y
120.50
Sum of X any Y is 376.25
Do you want me to do one more sum?
No Thanks
Bye then, See you again!
```

The lower-case words are of the computer and the words underlined are ours. Such interactive computing would be useful in determining certain intermediate results and taking actions depending upon the values.

## 2.9 PROBLEM SOLVING AND ALGORITHMS

Mathematical problems that can be solved through the computer may range in size and complexity. Since the computer does not possess any common sense and cannot make any unplanned decisions, the problem, whether it is simple or complex, has to be broken into a well-defined set of solution steps. It should be remembered that computers do not "solve" problems; rather, they are used to implement the solutions to problems.

In every instance of problem solving, the computer cannot be used to solve the problem until a method of solution has been evolved and a detailed procedure has been prepared by the user. It is assumed that the user has a certain amount of background knowledge, knows certain facts about the problem and possesses sufficient deductive and reasoning skills.

Problem solving involves the following steps:

1. studying the problem in detail
2. redefining or restating the problem
3. identifying output requirements, input data available and conditions and constraints to be used
4. comparing alternative methods of solution
5. selecting the method which is considered to be the best
6. preparing a logical and concise list of procedures or steps necessary for determining the solution
7. computing the results
8. examining the results for correctness

The computer's help may be necessary only in the seventh step. All the remaining steps are to be performed by the user. It is this fact that a beginner finds difficult to appreciate.

The logical and concise list of procedure for solving a problem is called an *algorithm*. It describes the steps that lead to unambiguous results in a finite number of operations. Figure 2.6 illustrates an algorithm for finding the square root of a set of *N* numbers.

Step 1: Find out the number of values for which square roots are to be evaluated.
Step 2: Take a value.
Step 3: See whether the value is positive or negative. If positive, go to *Step 4*, otherwise go to *Step 6*.
Step 4: Evaluate the square root.
Step 5: Record the value and its square root.
Step 6: Repeat Steps 2 to 5 until all the values are completed.

**Fig. 2.6** Algorithm for finding the square root of a given set of values

An algorithm prepared for the first time might need review to:

1. determine the correctness of various steps
2. reduce the number of steps, if necessary
3. increase the speed of solving the problem

An algorithm should also include steps to identify any abnormal data or results and take corrective measures, if possible. In case of large problems, we can break them into parts representing small tasks, prepare several algorithms and later combine them into one large algorithm. This is known as the *modular approach*.

Developing computer programs using the modular approach is known as *modular programming*. A module is a program unit or entity that is responsible for a single task. Modules (known as subprograms) are arranged in a hierarchical structure (similar to an organisation chart) as shown in Fig. 2.7. This is essentially *top-down design* in which bigger modules are broken into smaller ones such that they are small enough to be understood and easily coded using simple logic.

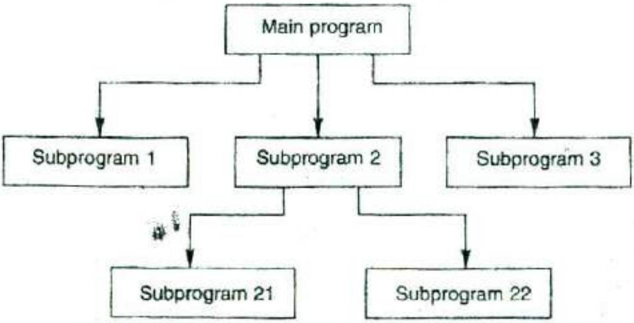


**Fig. 2.7** Top-down modular design of a program

## 2.10 FLOW CHARTING

When organising a problem for computer solution, it is desirable to present the algorithm pictorially. A flow chart is a diagram that outlines the sequences of operations to be performed. The operating steps are placed in boxes that are connected by arrows to indicate the order of execution of steps. Figure 2.8 illustrates the flow chart for the algorithm shown in Fig. 2.6. It is perhaps the best available method for expressing what the computer must do. Some symbols commonly used in flow charts are shown in Fig. 2.9.

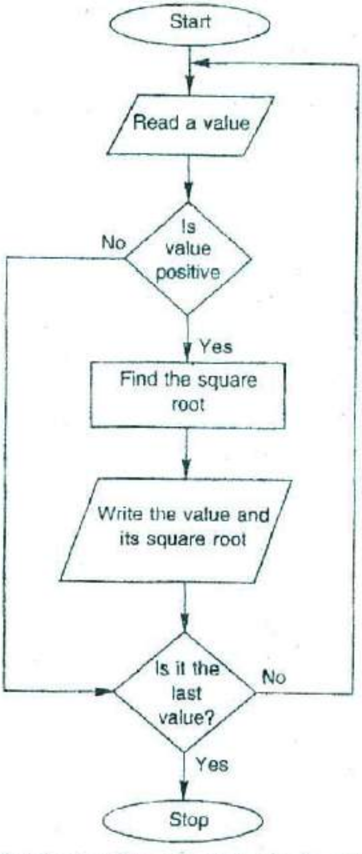


**Fig. 2.8** Flow chart for finding the square root of a given set of numbers

The important functions of a flow chart are as follows:

1. It provides a graphic representation of the problem so that it is easier to understand the plan of solution.
2. It provides a convenient aid to writing computer instructions (program).

3. It assists in reviewing and correcting the program.
4. It helps in discussion of the solution logic with others.

While drawing a flowchart, one must remember the following:

1. First list the logical steps.
2. Complete the main path of the logic first and then complete all branches and loops.
3. Use descriptive terms or mathematical equations in the boxes.
4. Each box should represent a step that is meaningful.
5. Use unambiguous terms in the flow chart so that others can easily understand it.

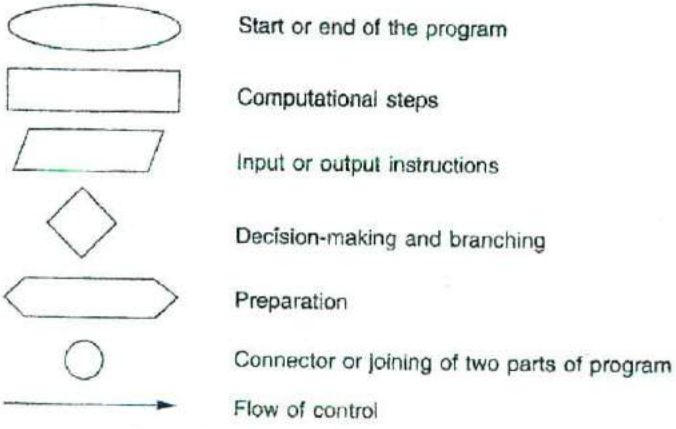

Start or end of the program
Computational steps
Input or output instructions
Decision-making and branching
Preparation
Connector or joining of two parts of program
Flow of control

**Fig. 2.9** Flow chart symbols

## 2.11 STRUCTURING THE LOGIC

Solution steps of all problems can be organised into one or combination of the following three control structures:

1. sequence structure
2. branching structure
3. looping structure

*Sequence* structure is used when the solution does not involve any repetitive operations or options. This is known as *straight-line logic* and is illustrated in Fig. 2.10.

*Branching* refers to the process of following one of two or more alternate paths of computations. This happens at a point where a test is performed to identify the conditions of certain variables in the process. The basis for selecting a particular path is stated within the decision box. The decision can be based on a comparison, on the value of a variable, on the sign of a variable, etc. The basic flow charts associated with branching are shown in Fig. 2.11.

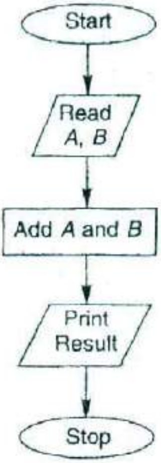


**Fig. 2.10** Sequence structure

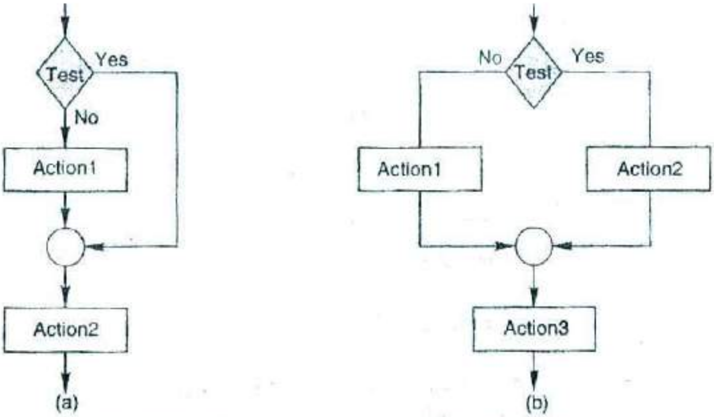
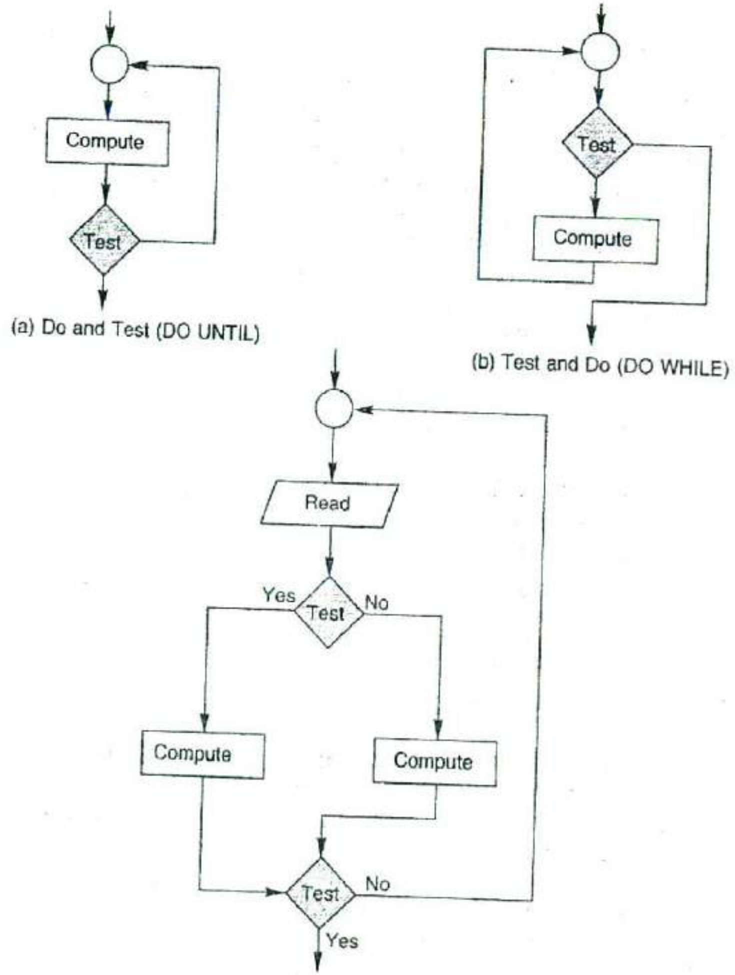


**Fig. 2.11** Branching structure (IF THEN ELSE)

In Fig. 2.11(a), a few steps are bypassed and the program is rejoined at a later stage. This is known as *forward jump*. In Fig. 2.11(b), each branch contains one or more computational steps. The two branches may join up again in the main path or may contain completely different steps and only join up at the end.

*Looping* refers to the repeated use of one or more steps. There are two types of loops. One is the *fixed loop* where the operations are repeated a fixed number of times. In this case, the values of the variables inside the loop have no effect on the number of times that the looping operation is performed. The other is the *variable loop* where the operations are repeated until a specified condition is met. Here, the number of times that the loop is repeated may vary. Searching for a particular item in a list of items is an example of variable loop.

Loops are also referred to as *backward jumps*. These jumps may occur either after meeting a specified condition in the process or after doing a certain computation. These jumps (loops) are illustrated in Fig. 2.12.



(a) Do and Test (DO UNTIL)

(b) Test and Do (DO WHILE)

(c) A mixed loop

Fig. 2.12 Illustration of loops

## 2.12 USING THE COMPUTER

Computers can be used to solve specific problems that may be scientific or commercial in nature. In either case, there are some basic steps involved in using the computers. These are as follows:

1. *Problem analysis* Identify the known and unknown parameters and state the constraints under which the problem is to be solved. Select a method of solution.
2. *Collecting information* Collect data, information and the documents necessary for solving the problem and also plan the layout of output results.
3. *Preparing the computer logic* Identify the sequence of operations to be performed in the process of solving the problem and plan the program logic, preferably using a program flow chart.
4. *Writing the computer program* Write the program of instructions for the computer in a suitable language.
5. *Testing the program* There may be errors (bugs) in the program. Remove all these errors which may be either in using the language or in the logic.
6. *Preparing the data* Prepare input data in the required form.
7. *Running the program* This may be done either in batch mode or interactive mode. The computations are performed by the computer and the results are given out.

The selection of a particular input/output device depends upon the nature of the problem, type of input data and the form of output required.

## 2.13 SUMMARY

We have discussed in this chapter the following aspects of computers and computing technology:

- evolution of computing devices
- generations of modern computers
- different types of computers
- input-process-output cycle of computing
- organisation and structure of a computer
- functions of various input, output and storage devices
- need for various types of computer programs
- importance of programming languages and their applications
- steps involved in solving mathematical problems
- use of flow charts for representing problem-solving algorithms
- application of modular and structured programming techniques for implementing computer-based solutions

### Key Terms

*Abacus*
*Algorithm*
*Analog computer*
*Application programs*
*Assembler*
*Assembly language*

*Low level language*
*Machine code*
*Machine language*
*Mainframe computer*
*Mark I*
*Microcomputer*

*(Contd.)*

*(Contd.)*

*Binary code*
*Branching structure*
*Compiler*
*Computer program*
*Computer-aided design*
*Computer-aided learning*
*Computer-aided manufacture*
*Computer-managed learning*
*Data*
*Debugging*
*Digital computer*
*ENIAC*
*Expert systems*
*Fifth generation*
*First generation*
*Floppy disk*
*Flow chart*
*Fourth generation*
*Hard disk*
*Hardware*
*High-level language*
*Hybrid computer*
*IC chips*
*Information*
*Interactive computing*
*Interpreter*
*Knowledge-based systems*
*Language processors*
*Large-scale integration (LSI)*
*Logarithm*
*Looping structure*
*Microelectronics*
*Microprocessor*
*Minicomputer*
*Modelling*
*Modular programming*
*Object program*
*Operation system*
*Parallel computer*
*PCAT*
*PCXT*
*Pentium*
*Personal computer*
*Second generation*
*Sequence structure*
*Simulation*
*Slide rule*
*Software*
*Source program*
*Straight-line logic*
*Structured programming*
*Supercomputer*
*Third generation*
*Top-down design*
*Transistor*
*Translators*
*UNIVAC*
*Utility programs*
*Vacuum tube*
*Winchester disk*
*Workstation*

## REVIEW QUESTIONS

1. Describe the abilities of modern computers that are directly relevant to numerical computing.
2. List at least two applications of computers in each of the following areas:
   (a) Industry
   (b) Business
   (c) Education
   (d) Engineering
3. Match the items in the following lists:
   (a) China (i) Punched Cards
   (b) John Napier (ii) Accounting Machine
   (c) Blaise Pascal (iii) Abacus

| | |
|---|---|
| (d) IBM | (iv) Logarithm |
| (e) Jacquard | (v) Mark I |

4. Describe briefly the developments in computing technology during the three decades from 1945 to 1975.
5. Describe the technology of fourth generation computers. How are they better than the earlier computer models?
6. What are fifth generation computers? How are they different from fourth generation systems?
7. Distinguish between analog and digital computers.
8. Distinguish between special purpose and general purpose computers.
9. A majority of computers used in the world today are digital. Why?
10. What are personal computers? How are they different from microcomputers?
11. Describe the relevance of supercomputers to engineers and scientists.
12. Define each of the following terms in one sentence:
    (a) Computer Program
    (b) Hardware
    (c) Information
    (d) Data
    (e) Software
13. Describe the functions of the following units in a computer:
    (a) Memory Unit
    (b) Arithmetic Logic Unit
    (c) Storage Unit
14. Describe how an application program is implemented in a computer.
15. Why do we need language processors? Describe the two forms of language processors available.
16. Compare the functions of application programs with that of operating systems.
17. What is machine language? What are its limitations?
18. How is assembly language better that machine language?
19. What are the features of high-level languages?
20. How is a program written in a high-level language implemented on a computer?
21. State the contributions of the following organisations to the development of high-level languages:
    (a) IBM
    (b) Bell Laboratory
    (c) US Defence Department
22. What are the advantages of interactive computing?
23. State the main steps involved in solving a mathematical problem.
24. What is an algorithm? How is it useful for a programmer?
25. What is modular programming? How does it help in solving a problem?
26. Why do we often use flow charts for developing computer programs?

27. Describe the three basic control structures used in executing the solution steps.
28. Critically compare the Do-and-Test and Test-and-Do looping structures.
29. Compare the following:
    (a) Forward jump versus backward jump
    (b) Fixed loop versus variable loop
30. Describe the basic tasks involved in solving a problem using a computer.

CHAPTER 3

# Computer Codes and Arithmetic

## 3.1 INTRODUCTION

Computers store and process numbers, letters and words that are often referred to as *data*. How do we communicate these numbers and words to computers? How do computers store this data and process them? Since computers cannot understand the Arabic numeral or English alphabet, we should use some "codes" that can easily be understood by them.

In all modern computers, storage and processing units are made of a set of silicon chips, each containing a large number of transistors. A transistor is a two-state device that can be put "off" and "on" by passing an electric current through it. Since the transistors are sensitive to currents and act like switches, we can communicate with the computers using electric signals, which are represented as a series of "pulse" and "no-pulse" conditions. For the sake of convenience and ease of use, a pulse is represented by the code "1" and a no-pulse by the code "0". They are called bits, an abbreviation of "binary digits". A series of 1's and 0's are used to represent a number or a character and thus, they provide a way for humans and computers to communicate with one another. This idea was suggested by John Von Neumann in 1946. The numbers represented by binary digits are known as *binary numbers*. Computers not only store numbers but also perform operations on them in binary form. Although information is stored in the computer memory in combinations of 0's and 1's, binary numbers become cumbersome when expressing large numbers. For this reason, internal contents of a computer are not displayed in binary form. Instead, they are displayed as *hexadecimal* or *octal* systems. Number systems that are popularly used in computing are the decimal system, binary system, hexadecimal system and octal system.

In this chapter, we will discuss the various number systems and their conversion from one system to another. We shall also discuss the internal representation of numbers and their arithmetic operations.

## 3.2 DECIMAL SYSTEM

The decimal number system, so familiar to us, is the oldest positional number system. In a positional system, a number is represented by a set of symbols. Each symbol represents a particular value depending on its position. The actual number of symbols used in a positional system depends on its *base*.

The decimal system uses a base of 10 and thus it uses 10 symbols, 0 to 9. Any number can be represented by arranging symbols in various positions. In the decimal system, each position represents a specific power of 10. Each successive position to the left of the decimal point represents a value ten times greater than the position to its immediate right as shown below :

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Position | ⟶ | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Place Value | ⟶ | $10^6$ | $10^5$ | $10^4$ | $10^3$ | $10^2$ | $10^1$ | $10^0$ |

For example, the decimal number 5704 represents:

| | | | | | |
|---|---|---|---|---|---|
| 3 | 2 | 1 | 0 | ⟵ | Position |
| $10^3$ | $10^2$ | $10^1$ | $10^0$ | ⟵ | Value |
| 5 | 7 | 0 | 4 | ⟵ | Decimal point |

$4 \times 10^0 = 4$

$0 \times 10^1 = 0$

$7 \times 10^2 = 700$

$5 \times 10^3 = 5000$

Sum 5704

We can express this in general form as

$$d_m\,(10^m) + d_{m-1}\,(10^{m-1}) + \ldots + d_0 = \sum_{i=0}^{m} d_i\ 10^i$$

where $d_i$ are the decimal symbols, 0 to 9 and $m - 1$ are the number of symbols. This is called the expanded notation for the integer.

Similarly, a fractional part of a decimal number can be represented as

$$\sum_{i=1}^{n} d_i\ 10^{-i}$$

where $n$ is the number of symbols in the fractional part.

## 3.3 BINARY SYSTEM

The binary system is the positional number system to the base 2. It uses two symbols 0 and 1. Again, each position in a binary number represents a power of the base as shown below.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Position | ⟶ | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Place Value | ⟶ | $2^6$<br>(64) | $2^5$<br>(32) | $2^4$<br>(16) | $2^3$<br>(8) | $2^2$<br>(4) | $2^1$<br>(2) | $2^0$<br>(1) |

Note that each successive position in the integer part of the binary number has a value two times greater than the position to its right.

For example, the binary number 1101 represents the decimal values as shown below:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 3 | 2 | 1 | 0 | ⟵ | Position | | |
| $2^3$ | $2^2$ | $2^1$ | $2^0$ | ⟵ | Value | | |
| 1 | 1 | 0 | 1 | ⟵ | Binary point | | |
| | | | └ | ⟶ | $1 \times 1$ | = | 1 |
| | | └ | | ⟶ | $0 \times 2$ | = | 0 |
| | └ | | | ⟶ | $1 \times 4$ | = | 4 |
| └ | | | | ⟶ | $1 \times 8$ | = | 8 |
| | | | | | | Sum | 13 |

That is, $1101_2 = 13_{10}$

The subscript 2 denotes a number in binary system and 10 denotes a number in the decimal system. In general form, it can be written as

$$d_m (2^m) + \mathrm{d}_{m-1} (2^{m-1}) + \ldots + d_0 = \sum_{i=0}^{m} d_i \, 2^i$$

where $d_i$ are the binary symbols, 0 or 1. We can further generalise the notation to any base $b$ as $\sum_{i \le m} d_i \, b^i$

Note that the base $b$ is usually an integer greater than one, and digits $d_i$ are between 0 and $b - 1$. The base is sometimes called *radix* and the fractional point is called *radix point*.

## 3.4 HEXADECIMAL SYSTEM

The hexadecimal system is a number system that uses 16 as its base. This system requires 16 one digit symbols. The first ten symbols are represented by digits 0 through 9 and the remaining six by the letters A through F. The letter A denotes 10, B denotes 11 and so on. Table 3.1 shows equivalents of decimal, binary, and hexadecimal values.

**Table 3.1** Equivalent values of different systems

| *Decimal System* | *Binary System* | | | | | *Hexadecimal System* |
|---|---|---|---|---|---|---|
| | *Weight* → | *8* | *4* | *2* | *1* | |
| 0 | | 0 | 0 | 0 | 0 | 0 |
| 1 | | 0 | 0 | 0 | 1 | 1 |
| 2 | | 0 | 0 | 1 | 0 | 2 |
| 3 | | 0 | 0 | 1 | 1 | 3 |
| 4 | | 0 | 1 | 0 | 0 | 4 |
| 5 | | 0 | 1 | 0 | 1 | 5 |
| 6 | | 0 | 1 | 1 | 0 | 6 |
| 7 | | 0 | 1 | 1 | 1 | 7 |
| 8 | | 1 | 0 | 0 | 0 | 8 |
| 9 | | 1 | 0 | 0 | 1 | 9 |
| 10 | | 1 | 0 | 1 | 0 | A |
| 11 | | 1 | 0 | 1 | 1 | B |
| 12 | | 1 | 1 | 0 | 0 | C |
| 13 | | 1 | 1 | 0 | 1 | D |
| 14 | | 1 | 1 | 1 | 0 | E |
| 15 | | 1 | 1 | 1 | 1 | F |

In the hexadecimal system, each position represents a value 16 times greater than the position to its immediate right. The place values of hexadecimal system are shown below:

| | | | | | | |
|---|---|---|---|---|---|---|
| Position | → | 4 | 3 | 2 | 1 | 0 |
| Place Value | → | $16^4$ | $16^3$ | $16^2$ | $16^1$ | $16^0$ |

The following example illustrates the decimal value represented by a hexadecimal number.

| | | | | | |
|---|---|---|---|---|---|
| 3 | 2 | 1 | 0 | ← | Position |
| $16^3$ | $16^2$ | $16^1$ | $16^0$ | ← | Value |
| 1 | 2 | A | F | ← | Hexadecimal point |

$15 \times 1 = 15$

$10 \times 16 = 160$

$2 \times 256 = 512$

$1 \times 4096 = 4096$

Sum 4783

Thus, $12AF_{16} = 4783_{10}$

To convert a binary number to hexadecimal, we need only to group the binary digits in sets of four and convert each group to its equivalent hexadecimal digit. Thus, the binary number 0111 1010 0001 0010 0001 becomes 7A121 in hexadecimal. This is illustrated below:

| | | | | | |
|---|---|---|---|---|---|
| Binary quadruplets | 0111 | 1010 | 0001 | 0010 | 0001 |
| | ↓ | ↓ | ↓ | ↓ | ↓ |
| Hexadecimal point | 7 | A | 1 | 2 | 1 |

This example clearly illustrates the advantage of hexadecimal system over binary system. For all large binary numbers, the hexadecimal representation is much more compact and, therefore, easier to write and manipulate than its binary equivalent.

## 3.5 OCTAL SYSTEM

The octal number system is a system having base $b$ as 8. The eight octal symbols are 0 through 7. The place values in the octal system are powers of 8 as shown below:

| | | | | | |
|---|---|---|---|---|---|
| Position ⟶ | 4 | 3 | 2 | 1 | 0 |
| Place Value ⟶ | $8^4$ | $8^3$ | $8^2$ | $8^1$ | $8^0$ |

The position values increase by a factor of 8 from right to left. The example below shows an octal number and its equivalent decimal value:

| | | | | |
|---|---|---|---|---|
| 3 | 2 | 1 | 0 | ⟵ Position |
| $8^3$ | $8^2$ | $8^1$ | $8^0$ | ⟵ Value |
| 2 | 0 | 5 | 6 | ⟵ Octal point |

$6 \times 1 = 6$

$5 \times 8 = 40$

$0 \times 64 = 0$

$2 \times 512 = 1024$

Sum 1070

Thus, $2056_8 = 1070_{10}$

Since $8 = 2^3$, each octal digit has a unique 3 bit binary representation. This is shown in Table 3.2.

**Table 3.2** Binary representation of octal digits

| *Octal* | *Binary Representation* |
|---|---|
| 0 | 000 |
| 1 | 001 |
| 2 | 010 |
| 3 | 011 |
| 4 | 100 |
| 5 | 101 |
| 6 | 110 |
| 7 | 111 |

Just as in the hexadecimal system, to convert a binary number to octal, it is only necessary to group the binary digits in sets of three and convert each set to its octal equivalent. For example, the binary number 1011010 can be represented in octal as follows:

| | | | |
|---|---|---|---|
| Binary triplets | 001 | 011 | 010 |
| | ↓ | ↓ | ↓ |
| Octal equivalent | 1 | 3 | 2 |

## 3.6 CONVERSION OF NUMBERS

We discuss here the following systems of conversion:

1. non-decimal to decimal system
2. decimal to non-decimal system
3. octal to hexadecimal system
4. hexadecimal to octal system

### Non-decimal System to Decimal System

We can convert a number in base 2, base 8 or base 16 to a decimal number using the expanded notation discussed so far. This conversion can also be accomplished using the following algorithms.

*Integral Part*

1. Multiply the *leftmost* digit by the base *b*
2. Add the next digit to the right to the product
3. Multiply the sum by the base *b* and add the next digit
4. Continue the process until the last (rightmost) digit is *added*

The sum is the decimal equivalent of the given integer number

*Fractional Part*

1. Multiply the *rightmost* digit by 1/*b*
2. Add the next digit to the left to the product
3. Multiply the sum by 1/*b* and add the next digit
4. Continue this process until the last (leftmost) digit in the fractional part is *added*
5. Multiply the last sum by 1/*b*

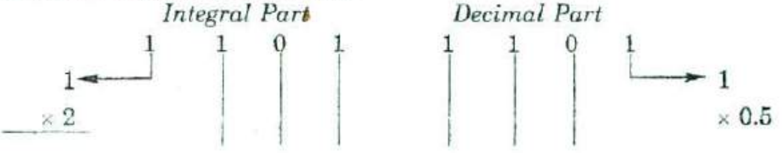
The product is the decimal equivalent of the given fractional number.

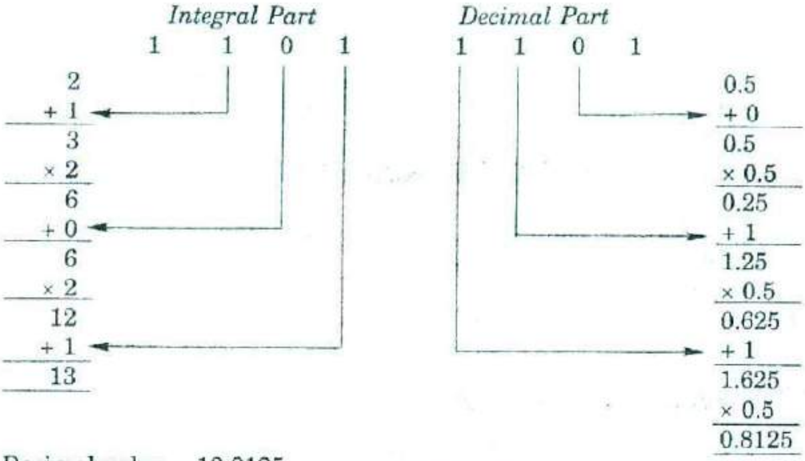
Note that, in the integral part algorithm, the process ends when the rightmost digit is added, but, in the case of fractional part algorithm, the process ends when the leftmost digit is added and the final sum is multiplied by 1/*b*.

#### Example 3.1

Convert binary number 1101.1101 to its decimal equivalent.

We can convert the given binary number to the decimal equivalent using the above algorithm as follows:



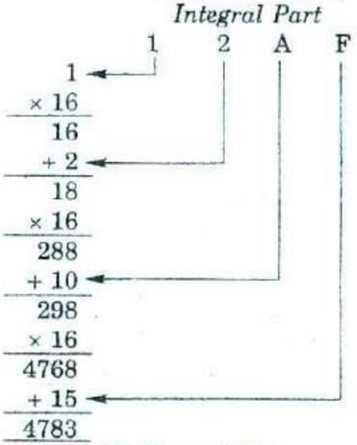


Decimal value = 13.8125

### Example 3.2

Convert the hexadecimal number 12AF to a decimal number

Conversion is done as follows:



Thus, $12AF_{16} = 4783_{10}$

## Decimal System to Non-decimal System

It is easy to convert a decimal number to a number of any other system. To do this, we must consider the integer and fractional parts separately as we did earlier. Algorithms to accomplish this are given below:

*Integral Part*

1. Divide the integer part of the decimal number by the base $b$ of the new system. The *remainder* will constitute the rightmost digit of the integer part of the new number.

2. Divide the quotient again by the base $b$. The remainder is the second digit from right.
3. Continue this process until a *zero quotient* is obtained. The last remainder is the leftmost digit of the new number.

*Fractional Part*

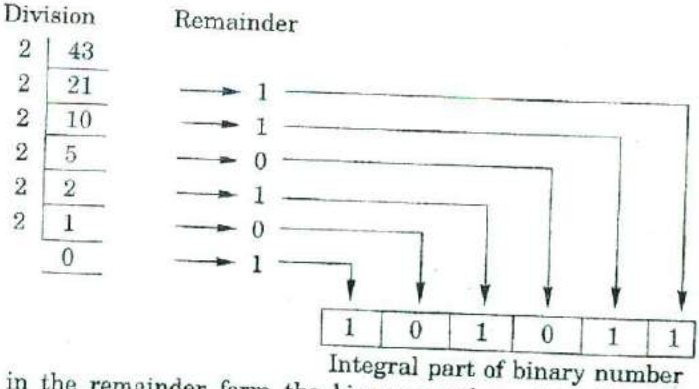
1. Multiply the fractional part of the decimal number by the base $b$ of the new system. The integral part of the product constitutes the leftmost digit of the fractional part of the new number.
2. Multiply the fractional part of the product by the base $b$. The integral part of the resultant product is the second digit from left.
3. Continue the process until a *zero fractional* part or a *duplicate fractional part* occurs. The integer part of the last product will be the rightmost digit of the fractional part of the new number.

Note that a duplicate fractional part indicates that the sequence will be an infinite one. The particular block of digits will be repeated over and over again.

**Example 3.3**

Convert the decimal 43.375 into its binary equivalent.

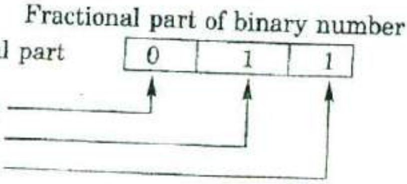
*Integral Part*



The digits in the remainder form the binary number when they are *dropped to the right.*

*Fractional Part*

| Multiplication | Product | Integral part |
|---|---|---|
| $0.375 \times 2$ | 0.75 | 0 |
| $0.750 \times 2$ | 1.50 | 1 |
| $0.500 \times 2$ | 1.00 | 1 |

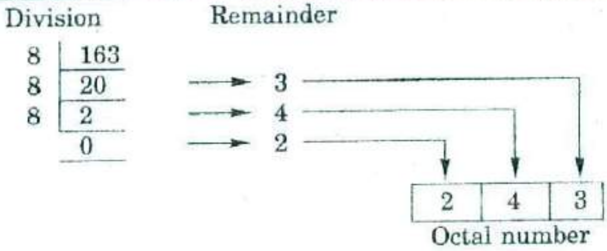


The digits in the integral part form the binary number when they are read from the top down (or *lifted up to the right* as shown).

Thus, $43.375_{10} = 101011.011_2$

**Example 3.4**

Convert the decimal number 163 to an octal number.



Thus, $163_{10} = 243_8$

**Example 3.5**

Convert the decimal number 0.65 to its binary equivalent.

| Multiplication | Product | Integral part |
|---|---|---|
| 0.65 × 2 | 1.3 | 1 |
| 0.3 × 2 | 0.6 | 0 |
| 0.6 × 2 | 1.2 | 1 |
| 0.2 × 2 | 0.4 | 0 |
| 0.4 × 2 | 0.8 | 0 |
| 0.8 × 2 | 1.6 | 1 |
| 0.6 × 2 | 1.2 | 1 |
| 0.2 × 2 | 0.4 | 0 |
| 0.4 × 2 | 0.8 | 0 |
| 0.8 × 2 | 1.6 | 1 |

Thus, $0.65_{10} = 1010011001$ ...

Note that a terminating decimal fraction need not have a terminating binary equivalent. This happens when a fractional part is repeated and therefore the process is terminated.

## Octal and Hexadecimal Conversion

Using the binary system as an intermediate stage, we can easily convert octal numbers to hexadecimal numbers and vice-versa. The steps are as follows:

*Octal to Hexadecimal*

1. Write the octal number.
2. Place the binary equivalent of each digit below the number.
3. Regroup them as binary quadruplets from the binary point, with zeros added, if necessary.
4. Convert each group into its equivalent hexadecimal digit.

*Hexadecimal to Octal*

1. Write the hexadecimal number.
2. Place the binary equivalent of each digit below the number.

3. Regroup them as binary triplets from the binary point, with zeros added if necessary.
4. Convert each group into its equivalent octal digit.

### Example 3.6

Convert the octal number 243 to a hexadecimal number.

| | | | | | |
|---|---|---|---|---|---|
| Octal | 2 | 4 | 3 | ← | Octal point |
| | ↓ | ↓ | ↓ | | |
| Binary equivalent | 010 | 100 | 011 | ← | Binary point |
| Regrouped as binary quadruplets | 0000 | 1010 | 0011 | | |
| | ↓ | ↓ | ↓ | | |
| Hexadecimal | 0 | A | 3 | | |

Thus, $243_8 = A3_{16}$

### Example 3.7

Convert the hexadecimal number 39.B8 to an octal number.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Hexadecimal | 3 | 9 | . | B | 8 | | |
| | ↓ | ↓ | | ↓ | ↓ | | |
| Binary equivalent | 0011 | 1001 | . | 1011 | 1000 | | |
| Regrouped as binary triplets | 000 | 111 | 001 | . | 101 | 110 | 000 |
| | ↓ | ↓ | ↓ | | ↓ | ↓ | ↓ |
| Hexadecimal | 0 | 7 | 1 | . | 5 | 6 | 0 |

Thus, $39.B8_{16} = 71.56_8$

The grouping of binary digits into triplets or quadruplets plays an important role in the internal organisation of information in computers. They are often used to represent long binary strings with lesser number of symbols.

## 3.7 REPRESENTATION OF NUMBERS

As mentioned earlier, all modern computers are designed to use binary digits to represent numbers and other information. The memory is usually organised into strings of bits called *words*. Each such string has the same length in a particular computer, although different computers may use different word lengths. For example, IBM PC and AT systems use a word length of 16 bits, while VAX 11 systems use a word length of 32 bits.

The largest number a computer can store depends on its word length. For example, the largest binary number a 16 bit word can hold is 16 bits of 1. This binary number is equivalent to a decimal value of 65535. The largest decimal number that can be stored in a computer is given by the following relation:

$$\text{Largest number} = 2^n - 1$$

where $n$ is the word length in bits. Thus, we see that the greater the number of bits, the larger the number that may be stored.

Although the computer works well with the binary numbers, humans do not. Firstly, it takes too many bits to represent a number. Secondly, writing such long series of bits can be exhausting and may cause errors. This is why we have other systems such as octal, hexadecimal and decimal systems. Computers read decimal numbers supplied by humans but convert them automatically into binary numbers for internal use. These binary numbers may also be expressed in the octal or hexadecimal form for print-out or display. For output, the numbers are reconverted to decimal form for human use.

## Integer Representation

Decimal numbers are first converted into the binary equivalent and then represented in either *integer* or *floating point* form. Let us first consider the integer representation.

For integers, the decimal or binary point is always fixed to the right of the least significant digit and therefore, fractions are not included. As mentioned earlier, the magnitude of the number is restricted to $2^n - 1$, where $n$ is the word length in bits.

How do we represent negative numbers? Negative numbers are stored by using the 2's complement. This is achieved by taking the 1's complement of the binary representation of the positive number and then adding 1 to it.

### Example 3.8

Represent $-13$ in binary form.

$$\begin{array}{rcr} 13 & = & 01101 \\ \text{1's complement} & = & 10010 \\ & & +\,00001 \\ \hline \text{2's complement} & = & 10011 \end{array}$$

Thus, $-13 = 10011$

Note that we have used an extra 0 to the left of the binary number representing 13. This is to indicate that the number is positive.

Then, if the leftmost bit is 1, the number is negative. The leftmost (or the most significant) bit of a binary number which is used to indicate the sign is called the *sign bit*.

Now we see that if we reserve one bit to represent the sign of the number, we have only $n - 1$ bits to represent the number. Thus, a 16 bit word can contain numbers $-2^{15}$ to $2^{15} - 1$ (i.e. $-32768$ to $32767$).

### Example 3.9

Show that the number –32768 is represented in a 16 bit word as follows:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

```
         –32768   =        (–32767) + (–1)
          32767   =     0111 1111 1111 1111
 1's complement   =     1000 0000 0000 0000
                  =   + 0000 0000 0000 0001
         –32767   =     1000 0000 0000 0001    ←——— (a)
              1   =     0000 0000 0001 0001
 1's complement   =     1111 1111 1111 1110
                  =   + 0000 0000 0000 0001
             –1   =     1111 1111 1111 1111    ←——— (b)
         –32768   =     1000 0000 0000 0000    ←——— (a) + (b)
```

## Floating Point Representation

We have just seen how integer numbers are represented. We have also seen that a 16 bit computer cannot store a positive number larger than 32767. What if we want to handle a fractional number like 35.7812 or a large number like 987654321? Such numbers are stored and processed in what is known as exponential form. These numbers have an embedded decimal point and are called *floating point numbers* or *real numbers*. For example, 35.7812 can be expressed 0.357812 × $10^2$. Similarly, the number 987654321 can be expressed as 0.987654 × $10^9$. By writing a large number in exponential form, we lose some digits. If $x$ is a real number, its floating point form representation is

$$x = f \times 10^E$$

The number $f$ is called *mantissa* and $E$ is the *exponent*.

Floating point numbers are stored differently. The entire memory location is divided into three fields or parts as shown in Fig. 3.1. The first part (1 bit) is reserved for the sign, the second part (7 bits) for the exponent of the number, and the third (24 bits) for the mantissa of the number. Typically, floating numbers use a field width of 32 bits where 24 bits are used for the mantissa and 7 bits for the exponent.

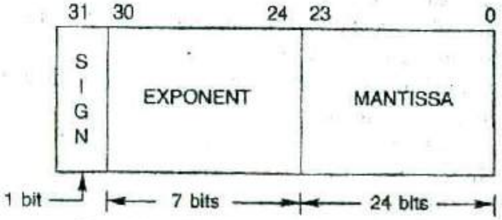


Fig. 3.1 Floating point representation

A-20696

Thus, we can represent very small fractions or very large numbers within the computer using the floating point representation.

### Example 3.10

Convert the following numbers to floating point notation.

0.00596, 65.7452, – 486.8

|  |  |  |
|---|---|---|
| 0.00596 | is expressed as | $0.596 \times 10^{-2}$ |
| 65.7452 | is expressed as | $0.657452 \times 10^{2}$ |
| –486.8 | is expressed as | $-0.4868 \times 10^{3}$ |

The shifting of the decimal point to the left of the most significant digit is called *normalisation* and the numbers represented in normalised form are known as *normalised floating point numbers*. You may note that the mantissa should satisfy the following conditions.

|  |  |
|---|---|
| For positive numbers: | less than 1.0 but greater than or equal to 0.1 |
| For negative numbers: | greater than –1.0 but less than or equal to –0.1 |

That is, $0.1 \leq |f| < 1$

The normalised floating point numbers are written using the following notation:

|  |  |  |
|---|---|---|
| $0.596 \times 10^{-2}$ | written as | .596 E – 2 |
| $-0.4868 \times 10^{3}$ | written as | – .4868 E3 |

## 3.8 COMPUTER ARITHMETIC

Different systems of computer arithmetic are currently available. They include integer arithmetic, fixed point arithmetic, floating point arithmetic, interval arithmetic and karlsruhe accurate arithmetic.

They are either supported by hardware or software or some software/hardware combinations. Each system uses its own scheme for representing numbers in binary form within the machine. The most common and popular arithmetic systems are integer arithmetic and floating point arithmetic. These systems are discussed briefly in this section.

### Integer Arithmetic

Virtually all computers offer integer arithmetic. The main property of integer arithmetic is that the result of any arithmetic operation with integers is an integer. The other property is that the result is always exact with the following two exceptions:

1. The range of integers that can be represented is not infinite but is bounded above and below.
2. The result of an integer division is usually given as a quotient and a remainder (since fractions cannot be represented in the integer scheme) which is truncated.

### Example 3.11

Illustrate integer arithmetic

| | |
|---|---|
| Addition: | $25 + 12 = 37$ |
| Subtraction: | $25 - 12 = 13$ |
| | $12 - 25 = -13$ |
| Multiplication: | $25 \times 12 = 300$ |
| Division: | $25 \div 12 = 2$ |
| | $12 \div 25 = 0$ |

Note that as only a finite range of integers can be represented, the product of two numbers may exceed the range.

### Example 3.12

Show that the following rule does not generally hold good in integer arithmetic.

$$\frac{a+b}{c} = \frac{a}{c} + \frac{b}{c}$$

Let $a = 5$, $b = 7$ and $c = 3$

Then, $$\frac{a+b}{c} = \frac{5+7}{3} = \frac{12}{3} = 4$$

$$\frac{a}{c} + \frac{b}{c} = \frac{5}{3} + \frac{7}{3} = 1 + 2 = 3$$

The results are not identical. This is because the remainder of an integer division is always truncated.

## Floating Point Arithmetic

Although integer arithmetic is adequate in many computing applications, it does not meet the requirements of many numerical computing methods as they often involve manipulation of fractional numbers. Hence, floating point arithmetic is the preferred choice for a majority of numerical computing applications.

In the floating point system, all the numbers are stored and processed in normalised exponential form. The most difficult operations in the floating point arithmetic are addition and subtraction.

***Addition*** Let the two numbers to be added be $x$ and $y$, and let $z$ be the result. Let the fractional parts and exponents be $f_x$, $f_y$ and $f_z$, and $E_x$, $E_y$ and $E_z$, respectively. Then, the addition algorithm is as follows:

1. Set $E_z$ = the larger of $E_x$ and $E_y$. (Assume here $E_x >= E_y$. Then $E_z = E_x$).
2. *Shift Right.* Shift $f_y$ to the right by $E_x - E_y$ places. (This makes the exponent of $f_x$ and $f_y$ the same).

3. *Add.* Set $f_z = f_x + f_y$
4. *Normalise.* If the absolute value of $f_z$ is greater than one, shift the decimal point of $f_z$ to the left of the most significant digit and then increase $E_z$ by one.

   Then $$z = f_z \times 10^{E_z}$$

In all the manipulations, the result of any operation is normalised and the mantissa is rounded or truncated to $p$ digits, where $p$ is the precision of the computer used. In the examples discussed here, we assume a precision of 6 and the mantissa is truncated to 6 digits.

### Example 3.13

Add the numbers 0.964572 E2 and 0.586351 E5.

Let $x$ = 0.586351 and $y$ = 0.964572 E2

$$E_z = 5$$
$$f_y = 0.000964$$
$$f_z = 0.000964 + 0.586351 = 0.587315$$

Then, $z$ = 0.587315 E5

Note that both the mantissa and exponent of the number with the smaller exponent are modified and the modified mantissa is truncated to six digits.

### Example 3.14

Add the numbers 0.735816 E4 and 0.635742 E4.

$$E_z = 4$$
$$f_z = 0.735816 + 0.635742 = 1.371558$$
$$z = 1.371558 \text{ E4} = 0.137155 \text{ E5}$$

Note that the mantissa of the result is truncated.

**Subtraction** Subtraction is nothing but addition of numbers with different signs. However, the subtraction of mantissas may result in a number less than 0.1. In such cases, the decimal point should be shifted to the left of the most significant digit and the exponent of the result should then be decreased accordingly.

### Example 3.15

Subtract 0.994576 E–3 from 0.999658 E–3.

Let $x$ = 0.999658 E–3 and $y$ = 0.994576 E–3

$$E_z = -3$$
$$f_z = 0.999658 - 0.994576$$
$$= 0.005082$$
$$z = x - y$$
$$= 0.005082 \text{ E} - 3$$
$$= 0.508200 \text{ E} - 5 \text{ (normalised)}$$

***Multiplication*** Multiplication of two floating point numbers is relatively simple.

1. Multiply the fractional parts: $f_z = f_x \times f_y$
2. Add the exponents: $E_z = E_x + E_y$
3. Then, $z = f_z \times 10^{E_z}$
4. Normalise, if necessary.

### Example 3.16

Multiply the numbers 0.200000 E4 and 0.400000 E – 2

$$\begin{aligned} f_z &= 0.200000 \times 0.400000 \\ &= 0.080000 \\ E_z &= 4 - 2 = 2 \\ z &= 0.080000\ \text{E2} = 0.800000\ \text{E1 (normalised)} = 0.800000 \end{aligned}$$

***Division*** Division is done as follows:

1. Divide the fractional parts: $f_z = f_x / f_y$
2. Subtract the exponents: $E_z = E_x - E_y$
3. $z = f_z \times 10^{E_z}$
4. Normalise, if necessary.

### Example 3.17

Divide the number 0.876543 E – 5 by 0.200000 E – 3

$$\begin{aligned} f_z &= 0.876543 \div 0.200000 \\ &= 4.382715 \\ E_z &= -5 - (-3) = -2 \\ z &= 4.382715\ \text{E} - 2 \\ &= 0.438271\ \text{E} - 1\ \text{(normalised)} \end{aligned}$$

Note that the mantissa of the result is truncated.

## 3.9 ERRORS IN ARITHMETIC

In integer arithmetic, while all arithmetic operations are exact, we might come across the following two situations:

1. An operation may result in a large number that is beyond the range of the numbers that the computer can handle.
2. An integer division may result in truncation of the remainder.

When the result is larger than the maximum limit, it is referred to as an *overflow* and when it is less than the lower limit, it is referred to as *underflow*. Unfortunately, most computers do not issue any warnings or messages on integer overflow or underflow. Therefore, we should use integer arithmetic with utmost care.

The floating point arithmetic system is prone to the following errors:

1. Error due to inexact representation of a decimal number in binary form. For example, consider the decimal number 0.1. The binary

equivalent of this number is 0.0001100110011.... The binary equivalent has a repeating fraction and therefore must be terminated at some point.

2. Error due to rounding method used by the computer, in order to limit the number of significant digits. This was illustrated in the examples discussed in Section 3.8. In fact, if the numbers added are too different in magnitude, the smaller may be treated as if it were zero (see Example 3.18).
3. Floating point subtraction may induce a special phenomenon. It is possible that some mantissa positions in the result are unspecified. This happens when two nearly equal numbers are subtracted. This is known as *subtractive cancellation*. If the operands themselves represent approximate values, the loss of significance is serious since it greatly reduces the number of significant digits. The error can be arbitrarily large (see Example 3.21).
4. Overflow or underflow can occur in floating point operations when the result is outside the limits of floating point number system of the computer.

The following examples illustrate these errors

### Example 3.18

Add the numbers 0.500000 E1 and 0.100000 E – 7.

Let

$$\begin{aligned} x &= 0.500000\text{ E1 and } y = 0.100000\text{ E} - 7 \\ E_z &= 1 \\ f_y &= 0.000000001 \\ f_z &= 0.500000001 = 0.500000 \\ z &= 0.500000\text{ E1} \end{aligned}$$

Note that the value of $z$ is the same as that of $x$.

### Example 3.19

Multiply the number 0.350000 E40 by 0.500000 E70

$$\begin{aligned} E_z &= 110 \\ f_z &= 0.175000 \\ z &= 0.175000\text{ E110} \end{aligned}$$

If we assume that the exponent can have a maximum value of 99, then the result *overflows*.

### Example 3.20

Divide the number 0.875000 E – 18 by 0.200000 E95.

$$E_z = -18 - 95 = -113$$

$$
\begin{aligned}
f_z &= 0.875000 \div 0.200000 = 4.375000 \\
z &= 4.375000 \text{ E} - 113 \\
&= 0.437500 \text{ E} - 114
\end{aligned}
$$

If we assume that the exponent can have a minimum value of – 99, then the result *underflows*.

### Example 3.21

Subtract 0.499998 from 0.500000.

$$
\begin{aligned}
f_x &= 0.500000 \\
f_y &= \underline{0.499998} \\
f_x - f_y &= \underline{0.000002} \\
E_z &= 0
\end{aligned}
$$

Thus, $z = 0.000002 \times 10^0 = 0.200000 \times 10^{-5}$

The result contains only one significant digit. If the values of *x* and *y* are not exact, then the result may not reflect the true difference between them. In many systems, these unspecified digits are filled by arbitrary digits thus causing a further increase in the error.

## 3.10 LAWS OF ARITHMETIC

Due to errors introduced in floating point arithmetic, the associative and distributive laws of arithmetic are not always satisfied. That is,

$$
\begin{aligned}
&x + (y + z) \neq (x + y) + z \\
&x \times (y \times z) \neq (x \times y) \times z \\
&x \times (y + z) \neq (x \times y) + (x \times z)
\end{aligned}
$$

Although failure of these laws to be satisfied affects relatively few computations, it can be very critical on some occasions. The examples that follow illustrate the discrepancies.

### Example 3.22

Associative law for addition

Let $x = 0.456732 \times 10^{-2}$, $y = 0.243451$, $z = -0.248000$

$$
\begin{aligned}
(x + y) &= 0.004567 + 0.243451 = 0.248018 \\
(x + y) + z &= 0.248018 - 0.248000 = 0.000018 \\
&= 0.180000 \times 10^{-4} \\
(y + z) &= 0.243451 - 0.248000 = -0.004549 = -0.4549 \times 10^{-2} \\
x + (y + z) &= (0.456732 - 0.454900)\ 10^{-2} \\
&= 0.183200 \times 10^{-2}
\end{aligned}
$$

Thus, $(x + y) + z \neq x + (y + z)$

### Example 3.23

Associative law for multiplication

Let $x = 0.400000 \times 10^{+40}$, $y = 0.500000 \times 10^{+70}$, $z = 0.300000 \times 10^{-30}$

$$(x \times y) \times z = (0.200000 \times 10^{+110})\,(0.300000 \times 10^{-30})$$

Note that $(x \times y)$ causes overflow and so the result will be erroneous.

$$x \times (y \times z) = (0.400000 \times 10^{40}) \times (0.1500000 \times 10^{40})$$
$$= 0.060000 \times 10^{80} = 0.600000 \times 10^{79}$$

This gives the correct result assuming that the exponent can take a value up to + 99.

### Example 3.24

Distributive law

Let $x = 0.400000 \times 10^{1}$, $y = 0.200001 \times 10^{0}$, $z = 0.200000 \times 10^{0}$

$$x \times (y - z) = (0.400000 \times 10^{1}) \times (0{,}100000 \times 10^{-5})$$
$$= 0.400000 \times 10^{-5}$$
$$(x \times y) - (x \times z) = 0.800000 \times 10^{0} - 0.800000 \times 10^{0} = 0$$

## 3.11 SUMMARY

In this chapter, we have discussed a very important aspect of numerical computing, namely, the internal representation of numbers in a computer. We considered the following in detail:

- number systems that are popularly used in computing
- conversion of numbers from one system to another
- storage of numbers in the memory of a computer
- different systems of arithmetic operations that are commonly used in numerical computing
- errors introduced by arithmetic operations
- associative and distributive laws of arithmetic

### Key Terms

*Associative law*
*Base*
*Binary digits*
*Binary numbers*
*Bits*
*Computer memory*
*Data*
*Decimal numbers*
*Distributive law*
*Interval arithmetic*
*Mantissa*
*Normalisation*
*Normalised numbers*
*Octal numbers*
*Overflow*
*Processing*
*Quadruplets*
*Radix point*

*(Contd.)*

*(Contd.)*

*Exponent*
*Fixed point arithmetic*
*Floating point form*
*Floating point arithmetic*
*Hexadecimal numbers*
*Integer arithmetic*
*Integer form*
*Real numbers*
*Sign bit*
*Storage*
*Subtractive cancellation*
*Transistor*
*Triplets*
*Underflow*

## REVIEW QUESTIONS

1. How many binary digits are there? Which symbols are used for them? What are they usually called?
2. Binary digits are used to store and manipulate data in computers. Why? Why do then we use other number systems?
3. What is the complement of a number? Obtain the complement of the decimal number 5749?
4. How do we obtain one's complement and two's complement of a binary number?
5. What are the uses of complements of binary numbers?
6. What is sign bit? How does the computer store a negative number?
7. An 8-bit register stores numbers in two's complement form.
   (a) What is the largest positive decimal number that can be stored?
   (b) What is the smallest negative decimal number that can be stored?
8. Why do we need to represent numbers in exponential form? Explain how a decimal number is represented inside the computer using the exponential form?
9. Explain the following:
   (a) Overflow
   (b) Underflow
10. Discuss the errors that may occur during the floating point arithmetic operations.
11. The hexadecimal equivalent of the binary number 10011101 is
    (a) 5A (b) FF (c) 9D (d) 9E
12. The decimal equivalent of the binary number 10011101 is
    (a) 27 (b) 157 (c) 13 (d) 144
13. The binary equivalent of the octal number 42 is
    (a) 101110 (b) 111010 (c) 100010 (d) 101011
14. The octal equivalent of the hexadecimal number CD5 is
    (a) 3625 (b) 6325 (c) 3652 (d) 6352
15. The hexadecimal equivalent of the decimal number 163 is
    (a) B3 (b) A2 (c) A3 (d) 93

## REVIEW EXERCISES

1. Convert the decimal numbers (i) 29, (ii) 123, and (iii) 432 to
   (a) binary system
   (b) octal system
   (c) hexadecimal system
2. Convert the octal numbers (i) 25, (ii) 52, and (iii) 563 to
   (a) decimal system
   (b) binary system
   (c) hexadecimal system
3. Convert the hexadecimal numbers (i) 8F, (ii) BC4, and (iii) AF3D to
   (a) decimal system
   (b) octal system
   (c) binary system
4. Convert the binary numbers (i) 0101, (ii) 0111.0111, and (iii) 1011.11 to
   (a) decimal system
   (b) octal system
   (c) hexadecimal system
5. Assuming that the computer stores each number in a 16-bit memory location, find the internal representations of the following numbers:
   (a) 498
   (b) –498
6. Write the following numbers in normalised exponential form and E-form.
   (a) 12.34 (d) –0.009876
   (b) –654.321 (e) 0.0
   (c) 0.001234 (f) 12345
7. Assuming that the mantissas are truncated to 4 decimal digits, show how the computer performs the following floating point operations:
   (a) $0.5678 \times 10^4 + 0.6666 \times 10^4$
   (b) $0.1234 \times 10^4 + 0.4455 \times 10^{-2}$
   (c) $0.3366 \times 10^{-2} - 0.2244 \times 10^{-1}$
   (d) $0.6789 \times 10^2 \times 0.2233 \times 10^{-1}$
   (e) $0.6789 \times 10^2 + 0.2233 \times 10^{-1}$
8. Assuming that the mantissas are truncated to 4 decimal digits, compute the error in the following computations:
   (a) 5.6789 – 1.2345
   (b) 5.6789 + 9.2345
9. Illustrate with examples the concept of overflow and underflow.
10. Discuss an example to show that the distributive law of arithmetic is not always satisfied in numerical computing.

CHAPTER 4

# Approximations and Errors in Computing

## 4.1 INTRODUCTION

Approximations and errors are an integral part of human life. They are everywhere and unavoidable. This is more so in the life of a computational scientist.

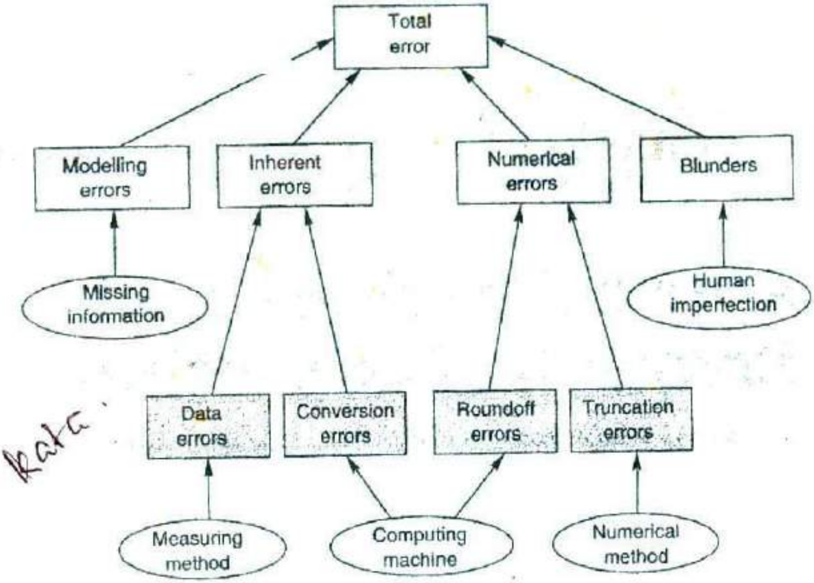
We cannot use numerical methods and ignore the existence of errors. Errors come in a variety of forms and sizes; some are avoidable, some are not. For example, data conversion and roundoff errors cannot be avoided, but a human error can be eliminated. Although certain errors cannot be eliminated completely, we must at least know the bounds of these errors to make use of our final solution. It is therefore essential to know how errors arise, how they grow during the numerical process, and how they affect the accuracy of a solution.

By careful analysis and proper design and implementation of algorithms, we can restrict their effect quite significantly.

As mentioned earlier, a number of different types of errors arise during the process of numerical computing. All these errors contribute to the total error in the final result. A taxonomy of errors encountered in a numerical process is given in Fig. 4.1 which shows that every stage of the numerical computing cycle contributes to the total error.

Although perfection is what we strive for, it is rarely achieved in practice due to a variety of factors. But that must not deter our attempts to achieve near perfection. Again the question is: How much near?

In this chapter we discuss the various forms of approximations and errors, their sources, how they propagate during the numerical process, and how they affect the result as well as the solution process.



Rata

**Fig. 4.1** Taxonomy of errors

## 4.2 SIGNIFICANT DIGITS

We know that all computers operate with a fixed length of numbers. In particular, we have seen that the floating point representation requires the mantissa to be of a specified number of digits. Some numbers cannot be represented exactly in a given number of decimal digits. For example, the quantity $\pi$ is equal to

$$3.1415926535897932384626...$$

Such numbers can never be represented accurately. We may write it as 3.14, 3.14159, or 3.141592653. In all cases we have omitted some digits.

Note that transcendental and irrational numbers do not have a terminating representation. Some rational numbers also have a repeating decimal pattern. For instance, the rational number 2/7 = 0.285714285714.... Suppose we write 2/7 as 0.285714 and $\pi$ as 3.14159. Then we say the numbers contain six *significant digits*.

The concept of significant digits has been introduced primarily to indicate the accuracy of a numerical value. For example, if, in the number $y = 23.40657$, only the digits 23406 are correct, then we may say that $y$ has five significant digits and is correct to only three decimal places.

In general, when a number is said to be "good to four digits", it means that the number has four significant digits. The omission of certain digits from a number results in what is called *roundoff error*. The following statements describe the notion of significant digits.

1. All non-zero digits are significant.
2. All zeros occurring between non-zero digits are significant digits.
3. Trailing zeros following a decimal point are significant. For example, 3.50, 65.0 and 0.230 have three significant digits each.
4. Zeros between the decimal point and preceding a non-zero digit are not significant. For example, the following numbers have four significant digits.

   0.0001234 $(1234 \times 10^{-7})$
   0.001234 $(1234 \times 10^{-6})$
   0.01234 $(1234 \times 10^{-5})$

5. When the decimal point is not written, trailing zeros are not considered to be significant. For example, 4500 may be written as $45 \times 10^2$ and contains only two significant digits. However, 4500.0 contains four significant digits. Further examples are:

   $7.56 \times 10^4$ has three significant digits.
   $7.560 \times 10^4$ has four significant digits.
   $7.5600 \times 10^4$ has five significant digits.

Integer numbers with trailing zeros may be written in scientific notation to specify the significant digits.

The concept of *accuracy* and *precision* are closely related to significant digits. They are related as follows:

1. Accuracy refers to the number of significant digits in a value. For example, the number 57.396 is accurate to five significant digits (sd).
2. Precision refers to the number of decimal positions, i.e. the order of magnitude of the last digit in a value. The number 57.396 has a precision of 0.001 or $10^{-3}$.

### Example 4.1

Which of the following numbers has the greatest precision

(a) 4.3201 (b) 4.32 (c) 4.320106

(a) 4.3201 has a precision of $10^{-4}$
(b) 4.32 has a precision of $10^{-2}$
(c) 4.320106 has a precision of $10^{-6}$

The last number has the greatest precision

### Example 4.2

What is the accuracy of the following numbers?

(a) 95.763 (b) 0.008472 (c) 0.0456000 (d) 36 (e) 3600 (f) 3600.00

(a) This has five sd.
(b) This has four sd. The leading or higher order zeros are only place holders.
(c) This has six sd.

(d) This has two sd.

(e) Accuracy is not specified.

(f) This has six sd. Note that the zeros were made significant by writing .00 after 3600.

## 4.3 INHERENT ERRORS

*Inherent errors* are those that are present in the data supplied to the model. Inherent errors (also known as *input errors*) contain two components, namely, *data errors* and *conversion errors*.

### Data Errors

Data error (also known as *empirical error*) arises when data for a problem are obtained by some experimental means and are, therefore, of limited accuracy and precision. This may be due to some limitations in instrumentation and reading, and therefore may be unavoidable. A physical measurement, such as a distance, a voltage, or a time period, cannot be exact. It is, therefore, important to remember that there is no use in performing arithmetic operations to, say, four decimal places when the original data themselves are only correct to two decimal places. For instance, the scale reading in a weighing machine may be accurate to only one decimal place.

### Conversion Errors

Conversion errors (also known as *representation errors*) arise due to the limitations of the computer to store the data exactly. We know that the floating point representation retains only a specified number of digits. The digits that are not retained constitute the roundoff error.

As we have already seen, many numbers cannot be represented exactly in a given number of decimal digits. In some cases a decimal number cannot be represented exactly in binary form. For example, the decimal number 0.1 has a non-terminating binary form like 0.00011001100110011.... but the computer retains only a specified number of bits. Thus, if we add 10 such numbers in a computer, the result will not be exactly 1.0 because of roundoff error during the conversion of 0.1 to binary form.

### Example 4.3

Represent the decimal numbers 0.1 and 0.4 in binary form with an accuracy of 8 binary digits. Add them and then convert the result back to the decimal form

$$0.1_{10} = 0.0001\ 1001$$
$$04_{10} = 0.0110\ 0110$$
$$\text{Sum} = 0.0111\ 1111$$

$$= 0.25 + 0.125 + 0.0625 + 0.03125 + 0.015625$$
$$+ 0.0078125 + 0.00390625$$
$$= 0.49609375$$

Note that the answer should be 0.5, but it is not. This is due to the error in conversion from decimal to binary form. Remember, both the numbers have non-terminating binary representation.

Error is equal to $2^{-8} = 0.00390625$. It is clear that the error can be reduced by increasing the binary digits that represent the number. For example, if we use 16 bits, then the error will be equal to $2^{-16} = 0.15258789 \times 10^{-4}$.

## 4.4 NUMERICAL ERRORS

*Numerical errors* (also known as *procedural errors*) are introduced during the process of implementation of a numerical method. They come in two forms, *roundoff errors* and *truncation errors*. The total numerical error is the summation of these two errors. The total error can be reduced by devising suitable techniques for implementing the solution. We shall see in this section the magnitude of these errors.

### Roundoff Errors

Roundoff errors occur when a fixed number of digits are used to represent exact numbers. Since the numbers are stored at every stage of computation, roundoff error is introduced at the end of every arithmetic operation. Consequently, eventhough an individual roundoff error could be very small, the cumulative effect of a series of computations can be very significant.

Rounding a number can be done in two ways. One is known as *chopping* and the other is known as *symmetric rounding*. Some systems use the chopping method while others use symmetric rounding.

### Chopping

In chopping, the extra digits are dropped. This is called *truncating* the number. Suppose we are using a computer with a fixed word length of four digits. Then a number like 42.7893 will be stored as 42.78, and the digits 93 will be dropped. We can express the number 42.7893 in floating point form as

$$x = 0.427893 \times 10^2$$
$$= (0.4278 + 0.000093) \times 10^2$$
$$= [0.4278 + (0.93 \times 10^{-4})] \times 10^2$$

This can be expressed in general form as

$$\text{True } x = (f_x + g_x \times 10^{\,d})10^E$$
$$= f_x \times 10^E + g_x \times 10^{E-d}$$
$$= \text{approximate } x + \text{error.}$$

where $f_x$ is the mantissa, $d$ is the length of the mantissa permitted and $E$ is the exponent. In chopping, $g_x$ is ignored entirely and therefore,

$$\text{Error} = g_x \times 10^{E-d},\ 0 \le g_x < 1$$

The absolute error introduced depends on the following:

1. the size of the digits dropped
2. number of digits in mantissa
3. the size of the number

Since the maximum value of $g_x$ is less than 1.0,

$$\boxed{\text{Absolute error} \le 10^{E-d}}$$

## Symmetric Roundoff

In the symmetric roundoff method, the last retained significant digit is "rounded up" by 1 if the first discarded digit is larger or equal to 5; otherwise, the last retained digit is unchanged. For example, the number 42.7893 would become 42.79 and the number 76.5432 would become 76.54.

As before, the value of unrounded number can be expressed as

$$\text{True } x = f_x \times 10^E + g_x \times 10^{E-d}$$

When $g_x < 0.5$, entire $g_x$ is truncated and therefore,

$$\text{Approximate } x = f_x \times 10^E$$

and

$$\text{Error} = g_x \times 10^{E-d}, \qquad g_x < 0.5$$

When $g_x \ge 0.5$, the last digit in the mantissa is increased by 1 and therefore

$$\text{Approximate } x = (f_x + 10^{-d}) \times 10^E = f_x \times 10^E + 10^{E-d}$$

$$\text{Error} = [f_x \times 10^E + g_x \times 10^{E-d}] - [f_x \times 10^E + 10^{E-d}]$$

$$= (g_x - 1) \times 10^{E-d} \qquad g_x \ge 0.5$$

In either case, $10^{E-d}$ is multiplied by factor whose absolute value is no greater than 0.5. Therefore, the value of the absolute error is

$$\boxed{\text{Absolute error} \le 0.5 \times 10^{E-d}}$$

Note that the symmetric rounding error is, at worst, one-half the chopping error.

Sometimes a slightly more refined rule is used when the $g_x$ is exactly equal to 0.5. Here $f_x$ is unchanged if its last digit is even and is increased by 1 if its last digit is odd.

### Example 4.4

Find the roundoff error in storing the number 752.6835 using a four digit mantissa.

$$\text{True } x = 0.7526 \times 10^3 + 0.835 \times 10^{-1}$$

*Chopping method*

$$\text{Approximate } x = 0.7526 \times 10^3$$
$$\text{Error} = 0.0835$$

*Symmetric rounding*

$$\text{Error} = (g_x - 1) \times 10^{-1}$$
$$= -0.165 \times 10^{-1} = -0.0165$$
$$\text{Approximate } x = 0.7527 \times 10^3$$

## Truncation Errors

Truncation errors arise from using an approximation in place of an exact mathematical procedure. Typically, it is the error resulting from the truncation of the numerical process. We often use some finite number of terms to estimate the sum of an infinite series. For example,

$$S = \sum_{i=0}^{\infty} a_i\, x^i \text{ is replaced by the finite sum } \sum_{i=0}^{n} a_i\, x^i$$

The series has been truncated.

Another example is the use of a number of discrete steps in the solution of a differential equation. The error introduced by such discrete approximations is also called *discretisation error*. Consider the following infinite series:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

When we calculate the sine of an angle using this series, we cannot use all the terms in the series for computation. We usually terminate the process after a certain term is calculated. The terms "truncated" introduce an error which is called *truncation error*.

Many of the iterative procedures used in numerical computing are infinite and, therefore, a knowledge of this error is important. Truncation error can be reduced by using a better numerical model which usually increases the number of arithmetic operations. For example, in numerical integration, the truncation error can be reduced by increasing the number of points at which the function is integrated. But care should be exercised to see that the roundoff error which is bound to increase due to increase in arithmetic operations does not off-set the reduction in truncation error.

We often use library functions to compute logarithms, exponentials, trigonometric functions, hyperbolic functions, and so on. In all these cases, a series is used to evaluate these functions. It is important to know the truncation errors introduced by these library functions. Truncation errors are discussed in detail in many places in this book.

### Example 4.5

Find the truncation error in the result of the following function for $x = 1/5$ when we use (a) first three terms, (b) first four terms, and (c) first five terms.

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \frac{x^6}{6!}$$

(a) Truncation error when first three terms are added

$$\text{Truncation error} = + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \frac{x^6}{6!}$$

$$= + \frac{0.2^3}{6} + \frac{0.2^4}{24} + \frac{0.2^5}{120} + \frac{0.2^6}{720}$$

$$= 0.1402755 \times 10^{-2}$$

(b) Truncation error when first four terms are added

Truncation error $= 0.694222 \times 10^{-4}$

(c) Truncation error when first five terms are added

Truncation error $= 0.275555 \times 10^{-5}$

### Example 4.6

Repeat the above example for $x = -1/5$

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \frac{x^6}{6!}$$

$$e^{-0.2} = 1 - 0.2 + \frac{0.2^2}{2} - \frac{0.2^3}{6} + \frac{0.2^4}{24} - \frac{0.2^5}{120} + \frac{0.2^6}{720}$$

(a) Truncation error (three terms) $= -\,0.1279255 \times 10^{-2}$

(b) Truncation error (four terms) $= +\,0.6665556 \times 10^{-4}$

(c) Truncation error (five terms) $= -\,0.257777 \times 10^{-5}$

Note that

$$|\text{T.E.}_3| < \frac{x^3}{3!}$$

$$|\text{T.E.}_4| < \frac{x^4}{4!}$$

$$|\text{T.E.}_5| < \frac{x^5}{5!}$$

## 4.5 MODELLING ERRORS

Mathematical models are the basis for numerical solutions. They are formulated to represent physical processes using certain parameters involved in the situations. In many situations, it is impractical or impossible to include all of the real problem and, therefore, certain simplifying assumptions are made. For example, while developing a model for calculating the force acting on a falling body, we may not be able to estimate the air resistance coefficient (*drag coefficient*) properly or determine the direction and magnitude of wind force acting on the body, and so on. To simplify the model, we may assume that the force due to air resistance is linearly proportional to the velocity of the falling body or we may assume that there is no wind force acting on the body. All such simplifications certainly result in errors in the output from such models.

Since a model is a basic input to the numerical process, no numerical method will provide adequate results if the model is erroneously conceived and formulated. It is obvious that we can reduce these type of errors by refining or enlarging the models by incorporating more features. But the enhancement may make the model more difficult to solve or may take more time to implement the solution process. It is also not always true that an enhanced model will provide better results. We must note that modelling, data quality and computation go hand in hand. An overly refined model with inaccurate data or an inadequate computer may not be meaningful. On the other hand, an oversimplified model may produce a result that is unacceptable. It is, therefore, necessary to strike a balance between the level of accuracy and the complexity of the model. A model must incorporate only those features that are essential to reduce the error to an acceptable level.

## 4.6 BLUNDERS

*Blunders* are errors that are caused due to human imperfection. As the name indicates, such errors may cause a very serious disaster in the result. Since these errors are due to human mistakes, it should be possible to avoid them to a large extent by acquiring a sound knowledge of all aspects of the problem as well as the numerical process.

Human errors can occur at any stage of the numerical processing cycle. Some common types of errors are:

1. lack of understanding of the problem
2. wrong assumptions
3. overlooking of some basic assumptions required for formulating the model
4. errors in deriving the mathematical equation or using a model that does not describe adequately the physical system under study

5. selecting a wrong numerical method for solving the mathematical model
6. selecting a wrong algorithm for implementing the numerical method
7. making mistakes in the computer program, such as testing a real number for zero and using < symbol in place of > symbol
8. mistakes in data input, such as misprints, giving values column-wise instead of row-wise to a matrix, forgetting a negative sign, etc.
9. wrong guessing of initial values

As mentioned earlier, all these mistakes can be avoided through a reasonable understanding of the problem and the numerical solution methods, and use of good programming techniques and tools.

## 4.7 ABSOLUTE AND RELATIVE ERRORS

Let us now consider some fundamental definitions of error analysis. Regardless of its source, an error is usually quantified in two different but related ways. One is known as *absolute error* and the other is called *relative error*.

Let us suppose that the *true value* of a data item is denoted by $x_t$ and its *approximate value* is denoted by $x_a$. Then, they are related as follows:

$$\text{True value } x_t = \text{Approximate value } x_a + \text{Error.}$$

The error is then given by

$$\text{Error} = x_t - x_a$$

The error may be negative or positive depending on the values of $x_t$ and $x_a$. In error analysis, what is important is the magnitude of the error and not the sign and, therefore, we normally consider what is known as *absolute error* which is denoted by

$$e_a = |x_t - x_a|$$

In many cases, absolute error may not reflect its influence correctly as it does not take into account the order of magnitude of the value under study. For example, an error of 1 gram is much more significant in the weight of a 10 gram gold chain than in the weight of a bag of rice. In view of this, we introduce the concept of *relative error* which is nothing but the "normalised" absolute error. The relative error is defined as follows:

$$e_r = \frac{\text{absolute error}}{|\text{true value}|}$$

$$= \frac{|x_t - x_a|}{|x_t|} = \left|1 - \frac{x_a}{x_t}\right|$$

More often, the quantity that is known to us is $x_a$ and, therefore, we can modify the above relation as follows:

$$e_r = \frac{x_t - x_a}{x_a} = \left|1 - \frac{x_t}{x_a}\right|$$

The fractional form of $e_r$ can also be expressed as the *per cent relative error* as

$$\text{Per cent } e_r = e_r \times 100$$

**Example 4.7** √ = |1 -

A civil engineer has measured the height of a 10 floor building as 2950 cm and the working height of each beam as 35 cm while the true values are 2945 cm and 30 cm, respectively. Compare their absolute and relative errors.

Absolute error in measuring the height of the building is

$$e_1 = 2950 - 2945 = 5 \text{ cm}$$

The relative error is

$$e_{r,1} = 5/2945 = 0.0017 = 0.17\%$$

Absolute error in measuring the height of the beam is

$$e_2 = 35 - 30 = 5 \text{ cm}$$

The relative error is

$$e_{r,2} = 5/30 = 0.17 = 17\%$$

Although the absolute errors are the same, the relative errors differ by 100 times. It shows that there is something wrong in the measurement of the height of the beam. It should be done more accurately.

## 4.8 MACHINE EPSILON

Recall that the round off error introduced in a number when it is represented in floating point form is given by

$$\text{Chopping error} = g \times 10^{E-d}, \qquad 0 \le g < 1$$

where $g$ represents the truncated part of the number in normalised form, $d$ is the number of digits permitted in the mantissa, and $E$ is the exponent. The absolute relative error due to chopping is then given by

$$e_r = \left|\frac{g \times 10^{E-d}}{f \times 10^{E}}\right|$$

The relative error is maximum when $g$ is maximum and $f$ is minimum. We know that the maximum possible value of $g$ is less than 1.0 and minimum possible value of $f$ is 0.1. The absolute value of the relative error therefore satisfies.

$$e_r \le \left|\frac{1.0 \times 10^{E-d}}{0.1 \times 10^{E}}\right| = 10^{-d+1}$$

The maximum relative error given above is known as *machine epsilon*. The name "machine" indicates that this value is machine dependent. This is true because the length of mantissa $d$ is machine dependent. For a decimal machine that uses chopping,

$$\text{Machine epsilon } \varepsilon = 10^{-d+1}$$

Similarly, for a machine which uses symmetric roundoff,

$$e_r \le \left|\frac{0.5 \times 10^{E-d}}{0.1 \times 10^{E}}\right| = \frac{1}{2} \times 10^{-d+1}$$

and therefore

$$\boxed{\text{Machine epsilon } \varepsilon = \frac{1}{2} \times 10^{-d+1}}$$

It is important to note that the machine epsilon represents upper bound for the roundoff error due to floating point representation. It also suggests that data can be represented in the machine with $d$ significant decimal digits and the relative error does not depend in any way on the size of the number.

More generally, for a number $x$ represented in a computer,

$$\boxed{\text{Absolute error bound} = |x| \times \varepsilon}$$

For a computer system with binary representation, the machine epsilon is given by

*Chopping*

Machine epsilon $\varepsilon = 2^{-d+1}$

*Symmetric rounding*

Machine epsilon $\varepsilon = 2^{-d}$

Note that we have simply replaced the base 10 by base 2. Here $d$ indicates the length of binary mantissa in bits.

We may generalise the expression for machine epsilon for a machine which uses base $b$ with $d$-digit mantissa as follows:

$\varepsilon = b \times b^{-d}$ for chopping

$\varepsilon = b/2 \times b^{-d}$ for symmetric rounding

### Example 4.8

When a computer uses a number base 2, how many significant decimal digits are contained in the mantissa of floating numbers?

Assume that the binary computer has $p$-bit mantissa. Then the error bound is $2^{-p}$. This computer will have $q$ significant digits with symmetric rounding, if,

$$2^{-p} = 1/2 \times 10^{-q+1}$$

Taking logarithms to the base 10, we get

$$q = 1 + (p - 1) \log_{10} 2$$

If we assume $p = 24$, then

$$q = 1 + 23 \log_{10} 2 \approx 7.9$$

We may say that the computer can store numbers with seven significant decimal digits.

## 4.9 ERROR PROPAGATION

Numerical computing involves a series of computations consisting of basic arithmetic operations. Therefore, it is not the individual roundoff errors that are important but the final error on the result. Our major concern is how an error at one point in the process *propagates* and how it effects the final total error. In this section, we will discuss the arithmetic of error propagation and its effects.

### Addition and Subtraction

Consider addition of two numbers, say, $x$ and $y$.

$$\begin{aligned} x_t + y_t &= x_a + e_x + y_a + e_y \\ &= (x_a + y_a) + (e_x + e_y) \end{aligned}$$

Therefore,

$$\boxed{\text{Total error} = e_{x+y} = e_x + e_y}$$

Similarly, for subtraction

$$\boxed{\text{Total error} = e_{x-y} = e_x - e_y}$$

Note that the addition $e_x + e_y$ does not mean that error will increase in all cases. It depends on the sign of individual errors. Similar is the case with subtractions.

Since we do not normally know the sign of errors, we can only estimate error bounds. That is, we can say that

$$|e_{x \pm y}| \leq |e_x| + |e_y|$$

Therefore, the rule for addition and subtraction is: *the magnitude of the absolute error of a sum (or difference) is equal to or less than the sum of the magnitudes of the absolute errors of the operands.*

This inequality is called the *triangle inequality*. The equality applies when the operands have the same signs, and the inequality applies if the signs are different.

### Multiplication

Here, we have

$$x_t \times y_t = (x_a + e_x) \times (y_a + e_y) = x_a y_a + y_a e_x + x_a e_y + e_x e_y$$

Errors are normally small and their products will be much smaller. Therefore, if we neglect the product of the errors, we get

$$\begin{aligned} x_t \times y_t &= x_a y_a + x_a e_y + y_a e_x \\ &= x_a y_a + x_a y_a \, (e_x/x_a + e_y/y_a) \end{aligned}$$

Then,

$$\boxed{\text{Total error} = e_{xy} = x_a y_a \, (e_x/x_a + e_y/y_a)}$$

## Division

We have

$$\frac{x_t}{y_t} = \frac{x_t + e_x}{y_a + e_y}$$

Multiplying both numerator and denominator by $y_a - e_y$ and rearranging the terms, we get

$$\frac{x_t}{y_t} = \frac{x_a y_a + y_a e_x - x_a e_y - e_x e_y}{y_a^2 - e_y^2}$$

Dropping all terms that involve only product of errors, we have

$$\frac{x_t}{y_t} = \frac{x_a y_a + y_a e_x - x_a e_y}{y_a^2}$$

$$= \frac{x_a}{y_a} + \frac{x_a}{y_a}\left(\frac{e_x}{x_a} - \frac{e_y}{y_a}\right)$$

Thus,

$$\boxed{\text{Total error} = e_{x/y} = \frac{x_a}{y_a}\left(\frac{e_x}{x_a} - \frac{e_y}{y_a}\right)}$$

Again applying the triangle inequality theorem, we have

$$e_{x/y} \le \left|\frac{x_a}{y_a}\right| \left(\left|\frac{e_x}{x_a}\right| + \left|\frac{e_y}{y_a}\right|\right)$$

$$e_{xy} \le |x_a y_a| \left(\left|\frac{e_x}{x_a}\right| + \left|\frac{e_y}{y_a}\right|\right)$$

*Note 1*

The initial errors $e_x$ and $e_y$ may be of any type. They may be

1. empirical errors introduced in the measuring process
2. roundoff errors introduced in conversion
3. roundoff errors introduced due to arithmetic operations in the previous step, if $x_t$ and $y_t$ represent some intermediate results

4. truncation errors, if $x_t$ and $y_t$ represent the result of evaluation of infinite series
5. any combination of the above

*Note 2*

The final errors (after arithmetic operations) $e_{x+y}$, $e_{x-y}$, $e_{xy}$ and $e_{x/y}$ are expressed in terms of only $e_x$ and $e_y$ and do not contain the roundoff errors introduced by the operations themselves. This results from the need to store the result in floating point representation. Therefore, we must add the roundoff error introduced in doing the operation in each case. For example,

$$e_{x+y} = e_x + e_y + e_o$$

Now, we can have relative errors for all the four operations as follows:

### *Addition and Subtraction*

$$e_{r,x\pm y} \le \frac{|e_x| + |e_y|}{|x_a \pm y_a|}$$

$$= \left|\frac{x_a}{x_a \pm y_a}\right| \cdot |e_{r,x}| + \left|\frac{y_a}{x_a \pm y_a}\right| \cdot |e_{r,y}|$$

### *Multiplication and Division*

$$e_{r,xy} = |e_{r,x}| + |e_{r,y}|$$

$$e_{r,x/y} = |e_{r,x}| + |e_{r,y}|$$

### Example 4.9

Estimate the relative error in $z = x - y$ when $x = 0.1234 \times 10^4$ and $y = 0.1232 \times 10^4$ as stored in a system with four-digit mantissa.

We know

$$e_{r,z} \le \frac{|e_x| + |e_y|}{|x - y|}$$

Since the numbers $x$ and $y$ are stored in a four-digit mantissa system, they are properly rounded off and therefore,

$$|e_{r,x}| \le \frac{1}{2} \times 10^{-3} = 0.05\%$$

$$|e_{r,y}| \le \frac{1}{2} \times 10^{-3} = 0.05\%$$

Then

$$e_x = 0.1234 \times 10^4 \times 0.5 \times 10^{-3} = 0.617$$

$$e_y = 0.1232 \times 10^4 \times 0.5 \times 10^{-3} = 0.616$$

Therefore

$$|e_z| \le |e_x| + |e_y| = 1.233$$

$$|e_{r,z}| \le \frac{1.233 \times 10^{-4}}{|0.1234 - 0.1232|} = 0.6165 = 61.65\%$$

Although the relative errors in $x$ and $y$ are very small, the relative error in $z$ is very large. If we use this result as an input to further calculations, the final result will be disastrous. The error due to subtraction of two nearly equal numbers is known as *subtractive cancellation*.

Rules for error propagation discussed above can also be derived using the concepts of differential calculus. We will find this approach more convenient when we deal with complex functions. For example, consider a power function

$$w = x^n$$

$$\text{Error } \Delta w = nx^{n-1}\,\Delta x$$

Relative error,

$$e_{r,w} = n \times \Delta x/x = n \times e_{r,x}$$

The relative error in $w$ is $n$ times the relative error in $x$.

## Sequence of Computations

We have seen how errors in the operands propagate to the result of an operation. As we know, the computer can do only one operation at a time. It performs a sequence of operations in order to evaluate even a simple expression; such as

$$w = x^2 + y/z$$

In such cases, the result of one operation is stored in the machine in the floating point form before it is used as an input for the next operation. At each stage of computation, a roundoff error is therefore introduced in the result before it is used again. Thus, each stage becomes a source of new errors. This is illustrated in Fig. 4.2, for evaluating the above expression. The intermediate value $u$ contains the propagated error due to error in $x$ and its own roundoff error $r_1$. Similarly, $v$ contains the propagated error due to errors in $y$ and $z$ and also the roundoff error $r_2$. Finally, $w$ contains the propagated error due to errors in $u$ and $v$ and the roundoff error $r_3$.

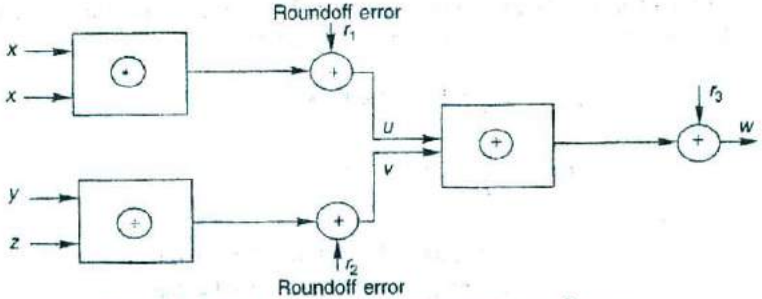


Fig. 4.2 Block diagram for evaluation of $x^2 + y/z$

**Example 4.10**

Find the absolute error in $w = xy + z$ if $x = 2.35$, $y = 6.74$ and $z = 3.45$

$$x_a = 2.35,\ e_x = 0.005\,|2.35| = 0.01175$$

$$y_a = 6.74,\ e_y = 0.005\,|6.74| = 0.03370$$

$$z_a = 3.45,\ e_z = 0.005\,|3.45| = 0.01725$$

$$e_{xy} = |x_a|e_y + |y_a|e_x$$

$$= 2.35 \times 0.03370 + 6.74 \times 0.01175 = 0.15839$$

$$e_w = |e_{xy}| + |e_z| = 0.15839 + 0.01725 = 0.17564$$

## Addition of a Chain of Numbers

As we pointed out earlier, many standard mathematical ideas do not hold good in computer arithmetic. One such case is the floating point addition. In computer arithmetic, the floating point addition is *not always associative*. That is,

$$x + y + z \neq z + y + x$$

The examples 4.11 and 4.12 illustrate this rule.

**Example 4.11**

Evaluate $w = x + y + z$, where $x = 9678$, $y = 678$ and $z = 78$. Remember, the computer performs arithmetic operations one at a time and from left to right. We assume that there is no inherent error (for the sake of simplicity) in $x$, $y$ and $z$ and the length of mantissa is four.

Let $u = x + y$

Then,

$$u = 0.9678 \times 10^4 + 0.0678 \times 10^4 = 1.0356 \times 10^4$$

$$w = u + z = 0.1035 \times 10^5 + 78$$

$$= 0.1035 \times 10^5 + 0.00078 \times 10^5$$

$$= 0.1042 \times 10^5 = 10420$$

True $w = 10444$

$$e_w = 24$$

$$e_{r,w} = 2.3 \times 10^{-3}$$

**Example 4.12**

Evaluate $w = z + y + x$ using the data in the above example.

Here, $u = 78 + 678 = (0.078 + 0.678)10^3 = 0.756 \times 10^3$

$$w = u + x$$

$$= 0.0756 \times 10^4 + 0.9678 \times 10^4 = 1.0434 \times 10^4$$

$$= 0.1043 \times 10^5 = 10430$$

$$\text{True } w = 10444$$

$$e_w = 14$$

$$e_{r,w} = 1.3 \times 10^{-3}$$

Examples 4.11 and 4.12 show that the errors are not the same in both the cases. It also shows that the error is less when the numbers are arranged in the increasing order of their magnitude. See Example 4.13 for a more general proof.

### Example 4.13

Prove that the procedure $w_1 = (y + z) + x$ is better than the procedure $w_2 = (x + y) + z$ when $|x| > |y| > |z|$.

*(a) Procedure* $w_2 = (x + y) + z$

Let $u = x + y$

Then,
$$e_{r,u} = \frac{x}{x+y} e_{r,x} + \frac{y}{x+y} e_{r,y} + r_1$$

where $r_1$ is the relative roundoff error introduced at this stage.

$$e_{r,w_2} = \frac{u}{u+z} e_{r,u} + \frac{z}{u+z} e_{r,z} + r_2$$

$$= \frac{x+y}{S}\left[\frac{x}{x+y} e_{r,x} + \frac{y}{x+y} e_{r,y} + r_1\right] + \frac{z}{S} e_{r,z} + r_2$$

$$= \frac{1}{S}\left[x \cdot e_{r,x} + y \cdot e_{r,y} + z \cdot e_{r,z} + (x+y)r_1 + (x+y+z)r_2\right]$$

where $S = x + y + z$

Now, let us use

$$R_1 = \max\left(|e_{r,x}|, |e_{r,y}|, |e_{r,z}|\right) \qquad R_2 = \max\left(|r_1|, |r_2|\right)$$

Then, we get

$$e_{r,w_2} = \frac{1}{S}\left[(x+y+z)R_1 + (2x+2y+z)R_2\right]$$

$$e_{w_2} = (x + y + z)R_1 + (2x + 2y + z)R_2$$

If we further assume that $R_1$ and $R_2$ are only due to conversion, then

$$R = R_1 = R_2 \; e_{w_2} = (3x + 3y + z)R$$

*(b) Procedure* $w_1 = (y + z) + x$

Similarly we can show that $e_{w_1} = (3z + 3y + x)R$

*Comparison*

$$e_{w_2} = (3x + 3y + z)R = (3x + 3y + 3z - 2z)R = (3S - 2z)R$$
$$e_{w_1} = (3x + 3y + 3z - 2x)R = (3S - 2x)R$$

Since $x > z$, $\qquad e_{w_1} < e_{w_2}$

Therefore, the procedure $w_1 = (y + z) + x$ gives better results than the procedure $w_2 = (x + y) + z$.

## Polynomial Functions

Suppose we wish to evaluate a function $f(x)$ where $f$ is differentiable and the approximate value $x_a$ of $x$ is given. In such cases, we can estimate the error bound in $f(x)$ using the *mean-value theorem* of calculus.

According to this theorem,

$$f(x) - f(x_a) = (x - x_a)f'(\theta)$$

where $\theta$ is some value between $x$ and $x_a$ and $f'$ is the first derivative of the function $f$. Then the error in $f(x)$ is

$$e_f = |f(x) - f(x_a)| = |e_x f'(\theta)|$$

Since the value of $\theta$ is unknown, we take the maximum of $f'(\theta)$ in the interval for estimating the bound for $e_f$. Then,

$$e_f \leq e_x \max |f'(\theta)|$$

This means that we have to evaluate the function $f'(\theta)$ at various values of $\theta$ and find the upper bound. This is sometimes a difficult task.

Normally, the error $e_x$ is small and, therefore, we can make a reasonable approximation as follows

$$\boxed{e_f \approx e_x f'(x_a)}$$

Note that this $e_f$ does not include the errors that occur during the evaluation of the function itself due to conversion at various stages.

### Example 4.14

Estimate the absolute and relative errors for the function

$$f(x) = \sqrt{x} + x \text{ for } x_a = 4.000$$

We assume that $x$ is correct to four significant digits. Then

$$e_x = 0.0005 = 5 \times 10^{-4}$$

$$f'(x_a) = \frac{1}{2} x^{-1/2} + 1 = \frac{1}{2}\sqrt{x} + 1$$

$$f'(x_a) = \frac{1}{4} + 1 = 1.25$$

Then

$$e_f = 5 \times 10^{-4} \times 1.25 = 6.25 \times 10^{-4}$$

$$e_{r,f} = \frac{e_r}{f(x_a)} = \frac{6.25 \times 10^{-4}}{6} = 0.104 \times 10^{-3}$$

The mean-value theorem approach can be readily extended to functions with more than one variable, using partial derivatives. For functions with two variables, $x$ and $y$, we have

$$e_f = \left|e_x f'_x\ (x_a, y_a)\right| + \left|e_y f'_y\ (x_a, y_a)\right|$$

where $f'_x$ and $f'_y$ denote partial derivatives with respect to $x$ and $y$.

### Example 4.15

Estimate the error in evaluating $f(x, y) = x^2 + y^2$ for $x = 3.00$ and $y = 4.00$

We assume that

$$e_x = e_y = 0.005$$

$$f'_x(x, y) = 2x \text{ and } f'_y(x, y) = 2y$$

Therefore,

$$e_f = 2x\, e_x + 2y\, e_y$$
$$= (2 \times 3.00 + 2 \times 4.00) \times 5 \times 10^{-3} = 0.07$$

## 4.10 CONDITIONING AND STABILITY

We know that uncertainties exist in all stages of numerical processing. We have discussed in detail how these uncertainties, particularly roundoff errors, are introduced at various stages and how they are propagated during the evaluation of an expression or implementation of a numerical method. Induced errors such as roundoff errors accumulate with the increasing number of computations in a process. There are situations where even a single operation may magnify the roundoff errors to a level that completely ruins the result. A computation process in which the cumulative effect of all input errors is grossly magnified is said to be *numerically unstable*. It is, therefore, important to understand the conditions under which the process is likely to be "sensitive" to input errors and becomes unstable. Investigations to see how small changes (or *perturbations*) in input parameters influence the output are termed as *sensitivity analysis*.

Numerical instability may arise due to sensitivity inherent in the problem or sensitivity of the numerical method (or algorithm). This is illustrated in Fig. 4.3. As we know, a mathematical model can be solved either by analytical methods or by numerical methods. In either case, when a small disturbance in an input parameter (known as *inherent error*) causes unacceptable amount of error in the output, we say that the problem is *inherently unstable*. Such problems are said to be *ill-conditioned*. When a problem itself is sensitive to small changes in its parameters, it is almost impossible to make a numerically stable method for its solution.

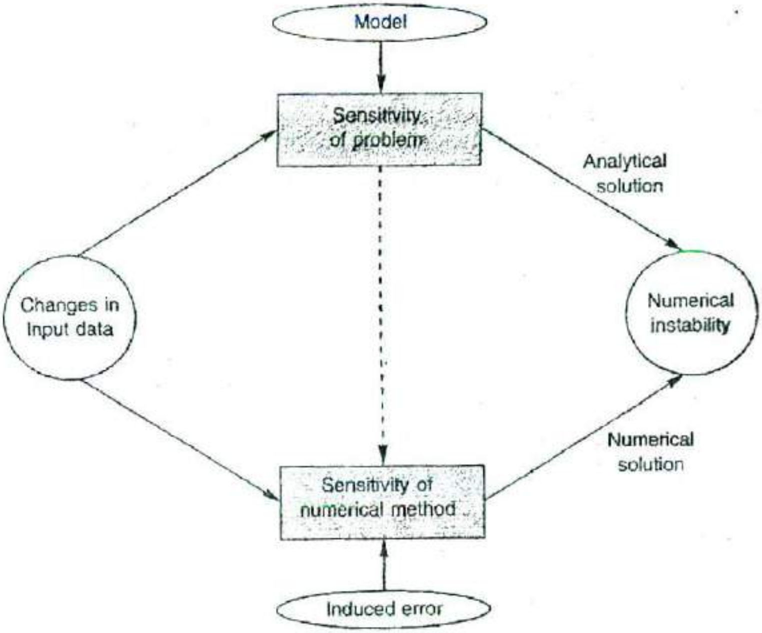


**Fig. 4.3** Instability of numerical process

The term "condition" is used to describe the sensitivity of problems or methods to uncertainty. Let us suppose we are evaluating a function $f(x)$ and a small change in $x$ produces a change in $f(x)$. We can quantify the condition of this function by a number called *condition number* which is defined as follows:

$$\text{Condition number} = \frac{\text{relative error in } f(x)}{\text{relative error in } x}$$

The relative error in $f(x)$ is

$$e_{r,f} = \frac{\Delta f}{f(x)} = \frac{f'(x)\Delta x}{f(x)}$$

The relative error in $x$ is

$$e_{r,x} = \frac{\Delta x}{x}$$

Then

$$\text{Condition number} = \frac{xf'(x)}{f(x)}$$

The condition number provides a measure of extent to which an error in $x$ is magnified in $f(x)$. If the condition number is large, then the function $f(x)$ is said to be ill-conditioned and its computation will be numerically unstable. There are different situations when a problem can have a large condition number.

1. small $f(x)$ compared to $x$ and $f'(x)$
2. large $f'(x)$ compared to $x$ and $f(x)$
3. large $x$ compared to $f(x)$ and $f'(x)$.

When several parameters are involved, we may have instability with respect to some parameters and stability with respect to others. In such cases, we should use the partial derivatives to estimate the total change. That is,

$$\Delta f = \left|\frac{\partial f}{\partial x}\Delta x\right| + \left|\frac{\partial f}{\partial y}\Delta y\right| + \left|\frac{\partial f}{\partial z}\Delta z\right| + \ldots$$

### Example 4.16

Show that the following system of equations is ill-conditioned for computing the point of intersection when $m_1$ and $m_2$ are nearly equal.

$$y = m_1x + C_1$$
$$y = m_2x + C_2$$

Solving the equations for $x$ and $y$ we get

$$x = \frac{C_1 - C_2}{m_2 - m_1}$$

$$y = m_1 \times \left[\frac{C_1 - C_2}{m_2 - m_1}\right] + C_1$$

Let us assume that $C_1 = 7.00$, $C_2 = 3.00$, $m_1 = 2.00$ and $m_2 = 2.01$. Then

$$x = \frac{7-3}{2.01 - 2.00} = 400$$

$$y = 2.00 \times 400 + 7 = 807$$

Now, let us change the value of $m_2$ from 2.01 to 2.005. Then

$$x = \frac{7-3}{2.005 - 2.00} = 800$$

$$y = 2.00 \times 800 + 7 = 1607$$

It shows that a small change (0.25 per cent) in the parameter $m_2$ results in almost 100 per cent change in the values of $x$ and $y$. Therefore, the problem is absolutely ill-conditioned.

### Example 4.17

Compute and interpret the condition number for

$$f(x) = \sqrt{(x-1)}$$

$$f'(x) = \frac{1}{2} \times (x-1)^{-1/2}$$

$$\text{Condition number} = \frac{xf'(x)}{f(x)} = \frac{x}{2}\frac{(x-1)^{-1/2}}{(x-1)^{1/2}}$$

$$= \frac{x}{2(x-1)}$$

The function is numerically unstable for the values of $x$ close to 1.

Note that the term "ill-conditioned" is ill-defined. If we are to take floating point seriously then we should say "relatively small changes" and "relatively large changes".

If the ill-conditioned effect is present in the original physical system itself, then there is nothing that we can do to achieve numerical stability. In many instances, the ill-conditioning arises from mathematical reformulation of the problem. In such cases, the instability may be removed by reformulating the mathematical models. For example, consider the quadratic equation

$$ax^2 + bx + c = 0.$$

We know that the two roots are

$$x_1 = \frac{-b+\sqrt{b^2-4ac}}{2a} \qquad x_2 = \frac{-b-\sqrt{b^2-4ac}}{2a}$$

When $b^2 >> 4ac, \sqrt{b^2-4ac}$ will be very close to $b$ and therefore, when $b$ is positive, the expression for $x_1$ may have the effect of *subtractive cancellation*. Here, we can reformulate the formula for $x_1$ as follows:

$$x_1 = \frac{-b+\sqrt{b^2-4ac}}{2a} \times \frac{-b-\sqrt{b^2-4ac}}{-b-\sqrt{b^2-4ac}}$$

$$= \frac{-2c}{b+\sqrt{b^2-4ac}}$$

If $b$ is negative, we must perform the same operation for $x_2$.

Another approach to the same problem is to change the algorithm of calculating $x_1$ and $x_2$. First find the larger root from the formula

$$x_1 = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$$

and then find the smaller root from the relation

$$x_1 x_2 = c/a$$

### Example 4.18

Compute the difference of square roots of two numbers $x = 497.0$ and $y = 496.0$.

Assume $x$ and $y$ are exact. Assuming a mantissa length of 4,

$$\sqrt{x} = \sqrt{497.0} = 0.2229 \times 10^2$$

$$\sqrt{y} = \sqrt{496.0} = 0.2227 \times 10^2$$

$$z = \sqrt{x} - \sqrt{y} = 0.0002 \times 10^2 = 0.02$$

Let us try another approach by rearranging the terms as follows:

$$z = \sqrt{x} - \sqrt{y} = \frac{x - y}{\sqrt{x} + \sqrt{y}}$$

$$= \frac{1}{0.4456 \times 10^2} = 0.2244 \times 10^1 = 0.02244$$

The correct answer is 0.02244. This shows that by rearranging the terms we improve the result.

### Example 4.19

Suggest an algorithm to compute the binomial co-efficient.

$$B = \frac{n!}{(n-r)!\, r!}$$

A simple algorithm to calculate $B$ is to find the factorials $n!$, $(n - r)!$ and $r!$ and combine them to get $B$. That is,

$$B = \frac{F_1}{F_2 \times F_3}$$

where $F_1 = n!$, $F_2 = (n - r)!$ and $F_3 = r!$

The problem with this algorithm is that when $n$ is large, the factorial $n!$ may be too large for the computer to store and thus, may result in overflow error. This problem can be overcome by modifying the algorithm as follows:

$$B = \left(\frac{n}{r}\right) = \left(\frac{n}{r-1}\right)\frac{n+1-r}{r}$$

$$B = n \text{ for } \quad r = 1$$

This can be expressed recursively as

$$B = n \prod_{i=2}^{r} \frac{n+1-i}{i}$$

This algorithm will compute $B$ without causing an overflow error unless the final answer itself is too large.

### Example 4.20

Reformulate the following expressions to avoid loss of accuracy due to subtractive cancellation.

(a) $x - \sqrt{x^2} - 1$ for large $x$

(b) $\dfrac{1 - \cos x}{\sin x}$ for small $x$

(a) $f(x) = x - \sqrt{x^2} - 1 = \dfrac{x^2 - (x^2 - 1)}{x + \sqrt{x^2} - 1} = \dfrac{1}{x + \sqrt{x^2} - 1}$

(b) $f(x) = \dfrac{1 - \cos x}{\sin x} = \dfrac{(1 - \cos x)(1 + \cos x)}{\sin x\,(1 + \cos x)}$

$$= \frac{\sin^2 x}{\sin x\,(1 + \cos x)} = \frac{\sin x}{1 + \cos x}$$

Even when the problem is formulated in a reasonable way and the input data is accurate, the method of solution may make the process unstable. For example, in a step-by-step algorithm where we use an interval $h$ to increment a variable, the error may increase if $h$ is decreased (or increased beyond some limit). If such *induced errors* are large, then our method of solution may exhibit what is known as *induced instability*. Another example is the "pivoting" technique used in solving simultaneous linear equations (see Chapter 7). Here, pivoting can make a well-conditioned system into an ill-conditioned one, if proper care is not taken in the design of algorithm.

### Example 4.21

Show that the series $e^x = 1 + x + \dfrac{x^2}{2!} + \dfrac{x^3}{3!} + \ldots +$

becomes unstable when $x = -10$.

The series can be represented as

$$S(x) = \sum_{i=0}^{n} T_i + T_E$$

where

$$T_i = \frac{x_i}{i!}$$

$T_E$ is the truncation error.

For $-1 < x < 1$, $T_i$ decreases as $i$ increases, but for large values of $|x|$, $T_i$ will grow in magnitude until the factorial in the denominator dominates, when once again $T_i$ will decrease in size. When $x = -10$, we have:

| | |
|---|---|
| $T_0$ | 1 |
| $T_1$ | −10 |
| $T_2$ | 50 |
| $T_3$ | −166.66767 |
| $T_4$ | 416.66767 |
| $T_5$ | −833.33333 |
| $T_6$ | 1388.8888... |

Assuming a six-digit mantissa machine, the roundoff errors in representing the large values of $T_i$ will be of greater magnitude than the final value $e^{-10} = 0.45 \times 10^{-4}$ itself. Therefore, the roundoff error totally dominates the computed solution. The method becomes unstable.

This above problem may be overcome by using a simple technique known as the range reduction scheme for $x$. We know that

$$e^x = (e^{x/2})^2$$

Thus,

$$e^{-10} = (e^{-1})^{10} = ((e^{-0.5})^2)^{10}$$

## 4.11 CONVERGENCE OF ITERATIVE PROCESSES

As pointed out earlier, most of the numerical computing processes are iterative in nature. We start with an approximate value of the solution and compute iteratively the next approximate value till the difference between two consecutive values is negligible or within a specified limit. The number of iterations required to reach the given limit depends on the rate at which the iterates converge to the result.

Suppose that $x_i$, $i = 0, 1, 2,...$ is a sequence of iterates and $x$ is the expected value of $x$. Let $e_i$ be the error in the iterate $x_i$. Then

$$e_i = x_i - x \qquad \text{for each } i$$

We would like the iterates to converge to $x$ and this would happen if the numerical process is stable. The process is said to converge if there exists positive constants $p$ and $c$ such that

$$\lim_{n \to \infty} \frac{|e_{i+1}|}{(|e_i|)^p} = c$$

The constant $p$ is known as the *order of convergence* and $c$ is known as *asymptotic convergence factor*. This shows that if the error in $x_{i+1}$ is proportional to the $p$th power of the error in $x_i$ (i.e. the previous iterate), then the iterative method is said to be of order of $p$. It is clear that the higher the order of iteration, more rapid is the *rate of convergence*.

The rate of convergence is a measure of how fast the truncation error goes to zero. This measure is used for comparing various iterative methods. The rate of convergence is expressed in different ways. For example, if the method converges like $h^2$, the order of convergence is $N^2$, and so on. It means the value of $p$ is 2. We shall consider the rate of convergence in detail when we discuss the iterative methods later.

## 4.12 ERROR ESTIMATION

It is now clear that it is almost impossible to know the exact error in a computed result. Nevertheless, it is possible at least to have some estimate of the error in the final result. There are three approaches that are popularly used in error estimation:

1. forward error analysis
2. backward error analysis
3. experimental error analysis

In *forward error analysis*, we try to estimate error bounds in the computed result using information such as uncertainties in the input data and the nature and number of arithmetic operations involved in the computing process. We can estimate the contribution due to

1. errors in the input data
2. roundoff errors in arithmetic operations
3. truncation of the iterative process
4. errors in formulation of the model

For example, we have seen in section 4.9 that the total error of a sum of three values is given by

$$e_s \le (x + y + z)R_1 + (2x + 2y + z)R_2$$

where

$$R_1 = \max(|e_{rx}|, |e_{ry}|, |e_{rz}|)$$
$$R_2 = \max(|r_1|, |r_2|)$$

This can be easily generalised for addition of $n$ values:

$$\begin{aligned} e_s &\le [(x_1 + x_2 + \dots + x_n)]R_1 + [(n-1)x_1 + (n-1)x_2 \\ &\quad + (n-2)x_3 + \dots + 2x_{n-1} + x_n]R_2 \\ &= R_1 \sum_{i=1}^{n} x_i + R_2 \left[(n-1)x_1 + \sum_{i=1}^{n-1} i \cdot x_{n-i+1}\right] \end{aligned}$$

Similarly, we can estimate bounds for product of $n$ numbers.

Error estimated through forward analysis is always pessimistic and is often much higher than the actual error.

In *backward error analysis*, we try to show that the computed results satisfy the problem within the given bounds. For example, we can put back the roots computed in the equation and see to what extent they satisfy the original equation. By comparison, we can then decide on how much confidence we can place in the computed results. Backward analysis is usually easier to perform than forward error analysis.

*Experimental error analysis* involves a series of experiments by using different methods and step sizes and then comparing the results. We may also perform *sensitivity analysis* to see how any change in parameters affects the result.

When the application is very critical in nature (such as space and defence applications) the problem may be solved by more than two independent specialists groups and the results can be compared.

## 4.13 MINIMISING THE TOTAL ERROR

Assuming that the mathematical model has been properly formulated and the input data are accurate, the total numerical error primarily consists of two components, namely, truncation and roundoff errors. Any effort to minimise the total error should, therefore, be concentrated on the ways to reduce these two types of errors. The steps may include:

1. increasing the significant figures of the computer
2. minimising the number of arithmetic operations
3. avoiding subtractive cancellations
4. choosing proper initial parameters

In many iterative processes such as numerical integration, it is possible to minimise the truncation error by decreasing the step size. But this would necessarily increase the number of iterations and thereby, arithmetic operations. This would certainly increase the roundoff error. This phenomenon is illustrated in Fig. 4.4. We must, therefore, judiciously choose a step size that would minimise the sum of these errors.

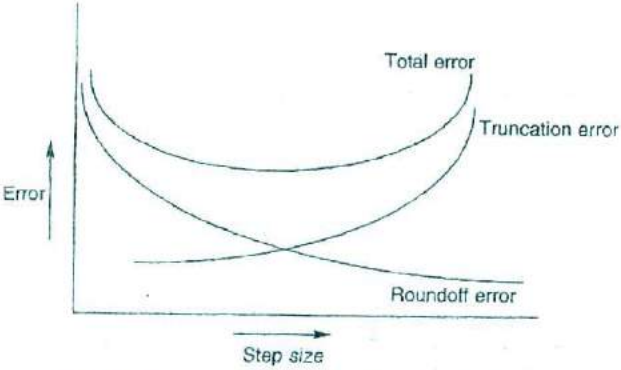


**Fig. 4.4** Dependence of error on step size

## 4.14 PITFALLS AND PRECAUTIONS

We have seen that the floating point arithmetic system is full of pitfalls such as conversion, roundoff, overflow and underflow errors. In many cases, we may have to consider some precaution techniques to get the most accurate results. The type of precaution techniques that might be used depends both on the computer hardware and the nature of the mathematical models. Here are some hints that might help improve the accuracy of the results.

1. Rearrange the formula so that you can avoid subtraction of two nearly equal numbers. For example,

   $$\frac{x^2 - y^2}{x - y}$$

   can be replaced by

   $$x + y$$

   when $x$ and $y$ are nearly equal.
2. If necessary, use double precision for floating point calculations. This would improve the accuracy considerably but would take more execution time and computer memory space.
3. Rearrange your formula to reduce the number of arithmetic operations. An example is evaluation of a polynomial. The polynomial

   $$a_n x^n + a_{n-1} x^{n-1} + \dots + a_0$$

   may be rearranged as

   $$(\dots((a_n x + a_{n-1})x + a_{n-1})x \dots + a_0)$$

   This requires much less arithmetic operations.
4. When finding the sum of set of numbers, arrange the set so that they are in the ascending order of absolute value. That is, when $|a| > |b| > |c|$, then $(c - b) + a$ is better than $(a - b) + c$.
5. Wherever possible, rearrange your formula so that you use the original data rather than derived data.
6. Do not test a floating point number for zero in your algorithm.
7. Wherever possible, use integer arithmetic to avoid conversion and roundoff errors.
8. Avoid multiplication of large numbers that may lead to overflow.
9. Use alternative arithmetic such as interval arithmetic, if necessary.

## 4.15 SUMMARY

In this chapter, we studied various types of errors and how they can affect numerical calculations. We considered, in particular, the following:

- concept of significant digits and its relation to accuracy and precision of numbers
- inherent errors that are present in input data
- procedural errors introduced during the process of computing
- modelling errors that arise due to certain simplifying assumptions in the formulation of mathematical models
- importance of absolute and relative errors and their relation to the machine epsilon

- propagation of errors during computing and how it affects the result
- causes of numerical instability and how to overcome instability problems
- convergence of iterative processes
- estimation of errors and some steps that might help to reduce the final error

### Key Terms

*Absolute error*
*Accuracy*
*Algorithm*
*Asymptotic convergence factor*
*Backward error analysis*
*Blunder*
*Chopping*
*Condition number*
*Conditioning*
*Convergence*
*Conversion error*
*Data error*
*Discretisation error*
*Drag coefficient*
*Empirical error*
*Error propagation*
*Experimental error analysis*
*Forward error analysis*
*Human error*
*Ill-conditioned problem*
*Induced errors*
*Induced instability*
*Inherent error*
*Inherently unstable*
*Input error*
*Machine epsilon*
*Mean-value theorem*
*Modelling error*
*Numerical error*
*Numerical instability*
*Numerically unstable*
*Order of convergence*
*Perturbations*
*Precision*
*Procedural error*
*Rate of convergence*
*Relative error*
*Representation error*
*Rounding*
*Roundoff error*
*Sensitivity analysis*
*Significant digits*
*Stability*
*Subtractive cancellation*
*Symmetric rounding*
*Triangle inequality*
*Truncating*
*Truncation error*

### REVIEW QUESTIONS

1. Why is the study of errors important to a computational scientist?
2. Explain the concept of significant digits.
3. Describe the relationship between significant digits and the following:
   (a) round-off errors
   (b) accuracy
   (c) precision
4. What are inherent errors? How do they arise?
5. Distinguish between roundoff errors and truncation errors.
6. What is chopping? When does it occur?

7. What is symmetric round-off? Show that the symmetric error is, at worst, one-half the chopping error.
8. How does a truncation error occur? Give two examples.
9. How do mathematical models contribute to errors in numerical computing?
10. What are blunders? How can we minimize them?
11. What do you mean by relative error? How is it important in error analysis?
12. What is machine epsilon? How is it related to significant digits?
13. State and explain triangular inequality as applied to error propagation.
14. What is subtractive cancellation? How does its presence affect the result of a computation?
15. Define condition number. What is its significance to numerical computing?
16. What is range reduction technique? Give an example of its application.
17. How will you decide the convergence of an iterative process?
18. Explain briefly the three approaches used in error analysis.
19. In an iterative process, how does step size affect the total error?
20. Enumerate a few precautionary steps that might help improve the accuracy of numerical computing.

## REVIEW EXERCISES

1. Find the accuracy and precision of the following numbers:

   (a) 12.345
   (b) 0.0002932
   (c) 0.0029320
   (d) 750
   (e) 750.5
   (f) –68.3705

2. Add the decimal numbers 0.4 and 0.65 in binary form using 6 binary digits and then estimate the error in the sum. Show that the error can be reduced by using more binary digits to represent the numbers.
3. Find the round-off error in the results of the following arithmetic operations, using four digit mantissa.

   (a) $27.65 + 22.20$
   (b) $87.26 + 31.42$
   (c) $1250.0 \times 40.0$
   (d) $3543.0 \times 16.78$
   (e) $25.68 \div 6.567$
   (f) $456.7 - 1.531$
   (g) $456.7 - 4.566$

4. Calculate absolute and relative errors in the arithmetic operations in Exercise 3.

5. Estimate the relative error of the final result in evaluation of
   (a) $w_1 = (x + y)z$ and
   (b) $w_2 = x^2 + y/z$

   Given that $x = 1.2$, $y = 25.6$ and $z = 4.5$.

6. Find the absolute and relative errors in evaluating the following expressions:
   (a) $\sqrt{x^2 + y^2}$
   (b) $x\,e^y$

   Assume $x = 1.25$ and $y = 2.16$.

7. Find out which procedure ($p_1$ or $p_2$) produces better results:
   (a) $p_1 = x(x + 2)$, $\quad p_2 = x^2 + 2x$
   (b) $p_1 = (x + 1)(x + 2)$, $\quad p_2 = x(x + 3) + 2$

8. Determine the condition of the following functions:
   (a) $f(x) = \sin(x)$
   (b) $f(x) = 1/(1 - x)$
   (c) $f(x) = x^5$
   (d) $f(x) = x^{1/3}$

9. Rearrange the following expression to avoid loss of accuracy due to subtractive cancellation:
   (a) $\cos x - \sin x$ for $x$ close to 45°
   (b) $\sqrt{1+x} - \sqrt{1-x}$ for small $x$
   (c) $1 - \cos x$ for small $x$
   (d) $\sqrt{x^2 + 1} - x$ for large $x$ .
   (e) $\ln(x + 1) - \ln(x)$ for large $x$

10. Estimate the maximum error in evaluating the expression

$$x^3 - 2.5x^2 + 3.1x - 1.5 \text{ at } x = 1.25$$

CHAPTER 5

# FORTRAN 77 Overview

## 5.1 NEED AND SCOPE

After a sound algorithm and a detailed flow chart comes the development of computer program, known as *coding*. Codes are written in a high-level computer language. Hundreds of high-level languages have been developed during the last four decades. Among these, a few have direct relevance to numerical computing. They include, among others, BASIC, FORTRAN, C, and C++.

FORTRAN, which stands for FORmula TRANslation, was the earliest scientific language developed in the 1950s. Since it was specially designed for mathematical computations, it has been the most widely used language for scientific and engineering applications. It is well suited for implementing the numerical methods discussed in this book. In spite of development of numerous other languages, FORTRAN continues to play a dominant role in engineering applications. Consequently, we are going to use FORTRAN for developing programs for implementing our algorithms. Our programs and algorithms are concise and general enough to be used as the basics for developing programs in other languages, if necessary.

A complete description of FORTRAN 77 is beyond the scope of this book. We only give here an overview of the language. However, enough material has been included so that the reader can easily understand the programs given in the book and also modify and implement them effectively. Wherever necessary, FORTRAN 90 features are also included.

## 5.2 A SAMPLE PROGRAM

For solving any problem in FORTRAN, we have to write a sequence of instructions using certain statements known as FORTRAN statements.

These instructions are required by the computer to perform the following tasks:

1. get data into the computer memory
2. perform arithmetic and logical operations on data
3. provide results on an output media

## *Program 5.1*

```
* -------------------------------------------------------- *
   PROGRAM SAMPLE
* -------------------------------------------------------- *
* Main program                                             *
*   A program to evaluate a function at different          *
    points                                                 *
* -------------------------------------------------------- *
* Functions invoked                                        *
*   NIL                                                    *
* -------------------------------------------------------- *
* Subroutines used                                         *
*   NIL                                                    *
* -------------------------------------------------------- *
* Variables used                                           *
*   X - Independent variable                               *
*   F - Function value                                     *
* COUNT - Counter to store number of evaluations           *
* -------------------------------------------------------- *
* Constants used                                           *
*   N - Number of function values                          *
* -------------------------------------------------------- *
      REAL X, F
      INTEGER COUNT, N
      PARAMETER( N = 5 )
      WRITE(*, *) 'Input value of X'
      READ(*, *) X
      WRITE(*, *) '  OUTPUT OF SAMPLE PROGRAM''
      WRITE(*, *) '        X          F         '
      COUNT = 0
100   F = X * X
      WRITE(*, *) X, F
      X = X + X
      COUNT = COUNT + 1
      IF( COUNT .LT. N ) GO TO 100
      STOP
      END
* -------------------------------------------------------- *
```

A sample FORTRAN 77 program to evaluate the function $f(x) = x^2$ for $n$ values of $x$ is shown in Program 5.1. When we run this program, it displays first the following message:

```
        Input Value of X
```

and then waits for the input from the keyboard. Let us enter a real value, say 1.0 and then press the RETURN key. Execution now continues and produces the following output on the screen:

```
Input value of X
1.0
    OUTPUT OF SAMPLE PROGRAM
         X              F
     1.0000000      1.0000000
     2.0000000      4.0000000
     4.0000000     16.0000000
     8.0000000     64.0000000
    16.0000000    256.0000000
Stop - Program terminated.
```

Program 5.1 illustrates some of the FORTRAN statements and the overall format of a FORTRAN 77 program. This program is intended to give only an overview of a FORTRAN program. The details of FORTRAN features will be discussed in the sections to follow.

The first line of Program 5.1 is a FORTRAN statement known as *program unit header* or *program statement*. This statement is not essential in all systems. You must consult the system manual before using it.

This is a recommended style in FORTRAN 90.

The lines starting with * or C in the first column are known as *comment lines* (only C in the FORTRAN IV version). These lines are used to insert explanatory remarks to help readers to understand the program. They are not instructions to the computer and, therefore, they are ignored by the compiler. Comment lines should be used liberally to explain various aspects within the program.

FORTRAN 90 permits the use of the character '!' in the first column to mark a comment line. This can also be used as an in-line comment.

The next two lines

```
    REAL X, F
    INTEGER COUNT, N
```

declare the types of storage associated with the variables. That is, the variables X and F are declared as type *real* and COUNT and N as type *integer*. These statements are called *type declaration* statements.

In FORTRAN 90, they are written as REAL:: X, F and INTEGER:: COUNT, N.

The identifier N represents the number of points at which the function is evaluated. The value of N is going to be constant throughout the program execution. Such identifiers are known as *symbolic constants* or *named constants*. Symbolic constants may be given values using a PARAMETER statement. (Some version of FORTRAN 77 may not include the feature of PARAMETER statement). The value of a symbolic constant cannot be changed during execution.

Some variables need to be given initial values like

```
COUNT = 0
```

before they are used in any expression. The process of setting variables to initial values is known as *initialisation*.

The set of statements

```
PRINT *, 'Input Value of X'
...
...
...
IF (COUNT .LT. N) GOTO 100
```

is known as *processing block*. It includes all *executable* statements such as input/output statements (READ, WRITE), assignment statements and control statements (such as IF). Note that the value of $x^2$ is evaluated and assigned to the variable F in this block. Similarly, the variables COUNT and X are incremented in this block. The statement

```
IF (COUNT .LT. N) GOTO 100
```

is known as a *control* statement.

This statement is responsible for creating a loop of operations and thereby making the function evaluated exactly N times. This is done with the help of the variable COUNT, usually known as a *counter*, which keeps counting the number of times the function has been evaluated.

Note that the statement

```
GOTO 100
```

directs the control to the statement

```
100 F = X * X
```

The number 100 is known as *statement number* or *statement label*. We need to use labels only to those statements to which the control is transferred from another part of the same program.

The last statement in our sample program is the END statement. This statement (which is a must in every FORTRAN program) serves two purposes:

1. it marks the end of source code during compilation
2. it terminates the execution of the program

> In earlier versions of FORTRAN, we need to use two statements:
>
> STOP — to stop the execution of the program
> END — to mark the physical end of the program.

FORTRAN 90 implements the same as follows:

```
    END  PROGRAM  SAMPLE
```

Note the structure of the sample program. A FORTRAN program generally consists of a series of blocks of code in the following order:

Program name
Program description
Variable declaration
Initialisation of symbolic constants
Initialisation of variables
Executable statements
The END statement

FORTRAN requires certain coding formats to be followed. Table 5.1 lists them.

**Table 5.1** FORTRAN 77 line format

| *Columns* | *Use* |
|---|---|
| 1 | For typing the comment character. |
| 1 - 5 | For typing statement number. |
| 6 | For typing a non-zero FORTRAN character to indicate that the previous statement is continued. |
| 7 - 72 | For typing FORTRAN statement. The statement can begin anywhere in the region. |
| 73 - 80 | Not used (or used for typing line number). |

FORTRAN supports the following major programming elements that have direct relevance to numerical computing discussed in this book.

1. constants
2. variables
3. input-output instructions
4. computational instructions
5. control instructions
6. documentation remarks
7. subprograms

The sample program has illustrated the use of all the first six elements. We shall discuss further details about them as well as the last element in this chapter.

## 5.3 FORTRAN CONSTANTS

Constants are the means by which numbers and characters are represented in a program. They are quantities that do not change. FORTRAN supports the following five built-in data types:

1. Integer type
2. Real type
3. Complex type

4. Logical type
5. Character type

*Integer constants* are numbers that do not contain decimal points (i.e. whole numbers). They can be positive, negative or zero. For examples

```
25    -10    0    +123
```

*Real constants* are numbers containing decimal points. They may be expressed in *positional form* or *exponential form*. Examples:

```
12.5        -1.756    0.0    5.    (Positional form)
.23 E+09    15E3      -2.3E -5     (Exponential form)
```

The exponential form (also known as *scientific form* or *floating point form*) is used where very large or very small numbers are to be written but not all digits need to be represented. (Number of significant digits depends on the computer.)

*Complex constants* are ordered pairs of real numbers, separated by commas and enclosed in parentheses, like (a, b). Examples:

```
(3.0, 4.0)    (-1.0, 0.92E2)    (1.2 E-2, 4.1 E1)
```

The first number is called the *real part* and the second is called the *imaginary part* of the complex number. (Complex numbers are usually written in $a + jb$ notation in mathematics.)

*Logical constants* are data that are used to represent the two truth values "true" and "false". Therefore, there are only two logical constants which are written as

```
.TRUE.
.FALSE.
```

*Character constants* represent a string of characters enclosed in apostrophes (single-quotes). Examples:

```
'John'    'January 26'    'NEW DELHI 20'    '123'
```

In FORTRAN 90, we may also use double quotes.

## 5.4 FORTRAN VARIABLES

Variables represent quantities that can change in value. In FORTRAN, they indicate storage locations where the values are stored. These values can be changed whenever required.

A variable name may consist of one to six characters, chosen from the letters A through Z and 0 through 9, the first of which must be a letter. Examples:

```
ALPHA    X1    SUM    NAME
```

FORTRAN 90 permits names with a length of 31 characters and also allows the use of underscore character.

We can have longer names in FORTRAN 90. Examples:

```
DISTANCE_TRAVELLED AVERAGE_HEIGHT
```

This facilitates to create more meaningful names.

All variables must be declared for their storage types corresponding to the five data types discussed in section 5.3, namely, integer, real, complex, logical and character. Examples:

```
REAL NUMBER, SUM, X1
INTEGER TOTAL, COUNT, Y
COMPLEX ROOT1, Z
LOGICAL PACKED, L
CHARACTER * 20 NAME, CITY
```

The variables NAME and CITY can hold up to 20 characters. We can also increase the number of significant digits held in a real type variable by declaring it a "double precision" variable as follows:

```
DOUBLE PRECISION SUM, X1
```

*Declaring a variable creates a storage location of appropriate type but it does not store any initial value. It contains some unknown bit pattern stored previously, i.e. the variable contains garbage.*

Any variable that is not declared explicitly for its type assumes default (*implicit*) type as follows:

| *Names beginning with* | *Type* |
|---|---|
| Any letter I through N | Integer |
| Any other letter | Real |

## 5.5 SUBSCRIPTED VARIABLES

FORTRAN variables can have subscripts to store a set of related values in one-dimensional vector or multidimensional matrices. A subscripted variable is called an *array*.

An array can be used to represent a collection of data of the same type and the subscripts can be used to access the individual data items. For instance, the third element of a one-dimensional array X is given by X(3). Examples of array variables are:

```
NAME(2, 10)    CITY(5)    GRADE(I)
```

We can use integer variables to represent subscripts and by assigning a suitable value to the subscript variables, we can access the desired element of the array.

All array variables must be declared for their type and size. Example:

```
REAL X(10)
```

or

```
REAL X(1:10)
```

Both these statements declare X as a one-dimensional array with elements numbered 1 to 10, which are of type real. The second form specifies the lower and upper bound of the subscript. In this form, the value of either bound may be positive, negative, or zero. The value of the upper bound should be greater than the value of the lower bound. Examples:

```
INTEGER  X(0:5), M(-10:20), N(-5:0)
REAL     P(5,5), VALUE(-3:3,5)
```

The second line declares P and VALUE as real type, two-dimensional arrays.

We may also declare type and size in separate statements like

```
REAL X, M
DIMENSION X(10), M(0: 10,10, 0:20)
```

Character arrays are declared as follows:

```
CHARACTER * 30 NAME(40)
```

or

```
CHARACTER * 30 NAME(1:40)
```

where the number 30 specifies the maximum number of characters to be stored in an array element.

In FORTRAN 90 arrays may be declared as follows:

```
REAL, DIMENSION(1:9) :: X,Y
CHARACTER(LEN = 30), DIMENSION(1:40) :: NAME
LOGICAL (-5:5) :: FOUND
CHARACTER, DIMENSION(10) :: CITY * 20
```

## 5.6 INPUT/OUTPUT STATEMENTS

Input/output statements are data transfer statements that are required in every program. FORTRAN supports two kinds of I/O statements

1. list-directed I/O statements
2. format directed I/O statements

We have already seen (in sample program 5.1) the use of list directed I/O statements. They are

```
READ *, X
PRINT *, F
```

The general form of these statements are:

READ *, $v_1$ , $v_2$ , ... , $v_n$
PRINT *, $v_1$ , $v_2$ , ... , $v_n$

where $v_1$, $v_2$, ..., $v_n$ are data items. READ * reads input data from the standard input device, usually the keyboard, and assigns them to the

**variables in the list.** PRINT * outputs the values of data items in the list on the standard output device, usually the screen. The data items $v_1$, $v_2$, ..., $v_n$ should be valid variables in case of READ and may be variables or constants in case of PRINT. Examples:

```
READ *, A, B, COUNT
PRINT *, X, 'TOTAL', SUM, 40.75
```

READ * is usually used to provide input data interactively through the keyboard. The data should be entered with either a comma or one or more spaces between the items.

Format directed I/O statements are used when the data should be read or written using a specified format. The general form of format directed I/O statements are

```
READ(n1, n2) v1, v2, ..., vn
WRITE(n1, n2) v1, v2, ..., vn
```

where $n_1$ is the number assigned to the device giving input or receiving output and $n_2$ is the number of the FORMAT statement which specifies the format of input data or output values. The FORMAT statement (a non-executable statement) takes the following form:

n2 FORMAT (list of specifications separated by commas)

**Examples:**

```
READ(5, 100) X, Y            (Reads values from unit 5)
WRITE(6, 200) X, Y, SUM      (Writes values to unit 6)
```

The unit may refer to keyboard, screen, printer, disk drive, and so on. If we are using only the standard devices as specified by the computer system, then we can use the following forms:

```
READ(* , 100) X, Y
WRITE(* , 200) X, Y, SUM
```

The FORMAT statements may look like

```
100 FORMAT (I5, F5.2)
200 FORMAT (I10, F7.2, F10.2)
```

The letter I indicates that the number to be handled is integer and F indicates that the number is floating point type. For more details about format specifications, you must consult the manual.

We may also give initial values to variables using the DATA statement as follows:

```
DATA X, Y / 25, 7.25 /
```

This statement assigns 25 to X and 7.25 to Y.

## 5.7 COMPUTATIONS

FORTRAN was specially designed to evaluate complex mathematical expressions. We can write a FORTRAN expression for a given mathematical expression and assign it to a variable using an assignment statement as follows:

$$\boxed{\text{v = expression}}$$

This statement directs the computer to replace the previous value of the variable on the left-hand side of the equals sign with the result of the expression on the right. The expressions are written using variables, constants and arithmetic operators (see examples shown in Table 5.2).

**Table 5.2** FORTRAN expressions

| *Algebraic expression* | *FORTRAN expression* |
|---|---|
| $a = x + \frac{y}{z} - r^2$ | `A = X + Y/Z - R ** 2` |
| $b = \frac{x+y}{z} + rt$ | `B = (X + Y)/Z + R * T` |
| $c = (xy)(z+2)$ | `C = (X * Y) * (Z + 2.0)` |

The following are the accepted arithmetic operators in FORTRAN:

| | |
|---|---|
| + | Addition |
| – | Subtraction or Unary minus |
| / | Division |
| * | Multiplication |
| ** | Exponentation |

According to the precedence rule

1. all exponents are performed first, all multiplications and divisions next, and all additions and subtractions last,
2. for the same precedence, the operations are performed from left to right, and
3. when parentheses are used, the expressions are evaluated from innermost to outermost parentheses (using the same precedence rule each time)

In numerical computations, we often come across an assignment statement of the type

```
SUM = SUM + N
```

This means, replace the "old value" of SUM by the "new value".

## Mixed-Mode Expressions

It is possible to combine integer, real and double precision quantities using these arithmetic operations. Expressions involving different types of numeric operands are called *mixed-mode expressions* and are evaluated as shown in Table 5.3.

**Table 5.3** Mixed-Mode Evaluation

| *Mixed-mode expression* | *Evaluation* | *Result type* |
|---|---|---|
| integer *op* real | Convert the integer to the corresponding real value and evaluate the expression. | Real |
| integer *op* double precision | Convert the integer to the corresponding double precision value and evaluate the expression. | Double precision |
| real *op* double precision | Extend the real to a double precision value (by adding zeros) and evaluate the expression. | Double precision |

## 5.8 CONTROL OF EXECUTION

Control of execution means the transfer of execution from one point to another in the same program, depending on the conditions of certain variables. This may involve a *forward jump* thus skipping a block of statements, or a *backward jump* thus repeating the execution of a block of statements. This is known as *conditional execution* of statements. Examples of such conditional execution are:

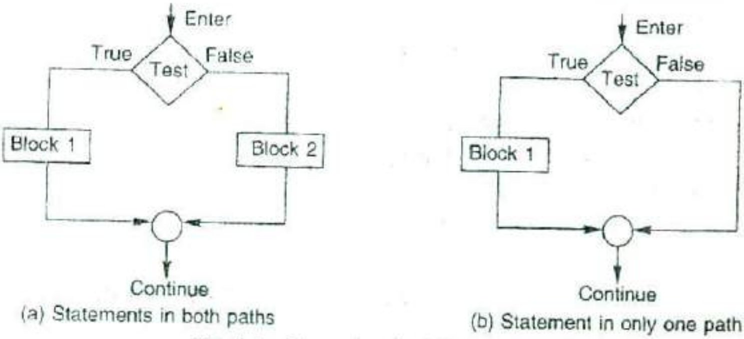
1. If the value is negative, skip the following four statements.
2. If the item is the last one, go to the end.
3. Execute the following ten lines 100 times.
4. Evaluate the following statement until a given condition is satisfied.

FORTRAN contains two central structures which could be used to implement such conditional execution of statements. They are

1. IF-ELSE structure
2. DO-WHILE structure

### Block IF-ELSE Structure

The block IF-ELSE structure (also known as *selection* structure) consists of a logical expression that tests for a condition or a relation followed by two alternative paths for the execution to follow. Depending on the test results, one of the paths is executed and the other is skipped. This is



(a) Statements in both paths

(b) Statement in only one path

**Fig. 5.1** Flow chart of IF-ELSE structure

illustrated in Fig. 5.1.

The FORTRAN statement to code a block IF-ELSE structure takes the form:

```
IF (logical expression) THEN ←
     statement-block 1                  IF block
ELSE                          ←
     statement-block 2                  ELSE block
END IF
```

The statement blocks may contain zero or more statements. If the logical expression is true, the program executes statement-block 1 and then goes to the statement next to the END IF statement; if the logical expression is false, the program executes statement-block 2 (skipping statement-block 1) and then goes to the statement next to the END IF.

## Relational Expressions

Relational expressions are meant for comparing the values of two arithmetic expressions and have logical values .TRUE. or .FALSE. as results. Arithmetic expressions may contain single variable, simple constant, intrinsic function, or a complex expression. In numerical computing, we often want our programs to test for certain relationships and make decisions based on the outcomes. We may use the *relational operators* given in Table 5.4 for comparing the expressions.

**Table 5.4** Relational operators

| *Operator* | *Meaning* |
|---|---|
| .LT. | Less than |
| .LE. | Less than or equal to |
| .EQ. | Equal to |
| .NE. | Not equal to |
| .GT. | Greater than |
| .GE. | Greater than or equal to |

Examples of rational operators are

```
1. IF(X .LT. Y) THEN
         PRINT * 'Small is', X
   ELSE
         PRINT * 'Small is', Y
   END IF
2. IF(TOTAL .GT. 1000) THEN
         TAX = 0.15 * TOTAL
   ELSE
         TAX = 0.10 * TOTAL
   END IF
         PRINT * 'GRAND_TOTAL = ', TOTAL + TAX
```

```
3. IF(C - D .GE. A - B) THEN
        X = C - D
   ELSE
        X = A - B
   END IF
```

> *When arithmetic expressions are used along with the relational operators, arithmetic expressions are evaluated first and then the results are compared.*

## Logical Expressions

In some cases, we may need to make more than one comparison. It is possible to combine two relational expressions using the following logical operators:

| | |
|---|---|
| `.AND.` | Both relations are true |
| `.OR.` | One or both of the relations are true |
| `.NOT.` | Opposite is true |

Such expressions are known as *logical expressions.*
Examples of logical expressions are:

```
1. IF(SUM .GT. 100 .OR. N .GT. 20) THEN
        ...
        ...
   ELSE
        ...
        ...
   END IF
2. IF(AGE .LT. 30 .AND. DEGREE .EQ. 'ME') THEN
        ...
        ...
   ELSE
        ...
        ...
   END IF
```

FORTRAN permits nesting of IF-ELSE blocks. That is, we can place an IF-THEN-ELSE code within an IF block or ELSE block.

> *Warning!*
>
> Be careful when comparing real values. They are never exact!

> We may also use the following relational operators in FORTRAN 90.
>
> < Less than
>
> <= Less than or equal to
>
> = = Equal to

/= Not equal to
>= Greater than or equal to
> Greater than

## DO-WHILE Structure

The DO-WHILE structure (also known as *looping* structure) performs a set of operations repeatedly while a certain condition is true. When the condition is not true, the repetition ceases. This kind of structure is implemented in FORTRAN by the DO statement. The general format of DO statement is:

```
DO n i = e1, e2, e3
      ...
      ...  <------------------ Body of the loop
      ...
n CONTINUE
```

where

n number of the last statement in the loop
i loop control variable
$e_1$ initial value of the control variable
$e_2$ final value of the control variable
$e_3$ increment value.

The control variable i may be a real or integer variable. The parameters $e_1$, $e_2$, and $e_3$ may be real or integer variables (or expressions or constants).

The default value of $e_3$ is 1. The logic of DO loop is as follows:

1. initialise the loop control variable to the initial value $e_1$
2. test to see if the value of loop control variable is less than or equal to the final value $e_2$. If it is true, continue the loop; otherwise exit the loop
3. execute the body of the loop
4. increment the loop control variable by $e_3$
5. go back to step 2 (beginning of the loop)

This can be written in pseudocode form as follows:

```
i = $e_1$
DO WHILE i <= $e_2$
    execute statements
    i = i + $e_3$
END DO
```

Figure 5.2 shows a flow chart showing the execution of the DO structure. The number of times the loop is executed (unless terminated by an EXIT statement) is given by the formula

$$m = \left\lfloor \frac{e_2 - e_1 + e_3}{e_3} \right\rfloor$$

$\lfloor x \rfloor$ denotes the greatest integer less than or equal to $x$.

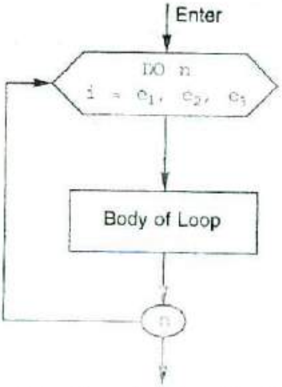


Fig. 5.2 Flow chart for the DO loop

Examples of DO loop are

```
1.      DO 10 P = X/Y, 75, Z/10.0
        ...
        ...
     10 CONTINUE
2.      DO 20 I = -4, 10, 0.25
        ...
        ...
     20 CONTINUE
3.      DO 30 N = 2, 20
        ...
        ...
     30 CONTINUE
4.      DO 40 J = 1, 100
        ...
        ...
        IF (...) GOTO 50    (Exit from the loop)
     40 CONTINUE
     50 ...
```

> *Warning !*
>
> Avoid the use of real variables for DO loop parameters. They cause roundoff errors and, therefore, cannot always guarantee the correct number of loop executions.

A DO loop can contain DO loops within its range. This is known as *nesting*. When nesting DO loops, the inner loop must be entirely contained within the range of the outer loop.

Examples of nesting DO loops are

```
1.    DO 200 I = 1, 10                 outer loop
         DO 100 J = 1, 5
         PRINT *, I * J                    inner loop
  100    CONTINUE
  200 CONTINUE

2.    DO 500 I = 1, 10
         DO 200 J = 1, 10              outer loop
         ...
         ...
  200    CONTINUE
     ...                                 inner loops
     ...
         DO 300 K = 1,5
         ...
         ...
  300    CONTINUE
  500 CONTINUE
```

The general form of DO structure in FORTRAN 90 is:

```
DO  loop control
    block of statements
END DO
```

This is implemented in two forms:

*Form 1*

```
DO i = e1, e2, e3
      ...
      ...
END DO
```

*Form 2*

```
1.  DO
        ...
        ...
        IF (...) EXIT
        ...
    END DO                        Leave the loop
    ...
2.  DO
        ...
        ...                       Go to the beginning
        IF (...) CYCLE
        ...
        ...
    END DO
```

## 5.9 SUBPROGRAMS

One of the features of any modern programming language is the provision for *subprograms*. A subprogram is a separate program unit that can be called into operation by other programs. Subprograms are heavily used in numerical computing for tasks such as evaluation of a function, matrix multiplication, sorting, reading a table of values, printing a report, etc.

The concept of subprograms allows us to break a complex problem into subtasks so that we may develop subprograms and later integrate them into a single program known as *driver* or *main* program. These subprograms can be independently designed, coded, and tested. Subprograms are usually called *modules* and the programming approach using modules is called *modular programming*.

FORTRAN supports two kinds of subprograms, namely, *functions* and *subroutines*. A function subprogram returns a single value to the calling program while a subroutine subprogram can compute and return several values.

### Function Subprograms

A function subprogram (or simply a function) is an independent program unit written to compute and return a single value. It takes the following form:

```
type FUNCTION name (arguments)
     Declaration of argument types
     ...
     ...  <------------------------- Execution statements
     ...
     name = expression
     RETURN
     END
```

where *type* specifies the type of the function value that is being returned and *arguments* are *dummy* variables that must be declared for their type inside the function. They may vary in number from zero to many. There should be at least one statement of the form

```
name = expression
```

which assigns a value of appropriate type to the function name, which is in turn returned to the calling program.

A function can be called as follows:

```
variable = name (arguments)
```

When the function is called, the values of the arguments in the calling statement are assigned to the corresponding arguments in the function header. The arguments, therefore, must agree in order, number and type. An argument may be a variable name, an array name, or a subprogram name. Example:

```
PROGRAM MAIN
REAL A, B, R, MUL  ←———  MUL declared in main
READ * A, B
R = MUL(A, B)      ←———  Calling MUL
PRINT * , R
END
_____________________________
REAL FUNCTION MUL(X, Y) ←———  MUL defined
REAL X, Y
MUL = X * Y
RETURN
END
```

When an array is passed as an argument, then its corresponding dummy argument should be an array variable and its size must be declared properly. Note that a function may be called and used in an expression, like any other variable. Example:

```
R = A * MUL (A, B)
```

## Subroutine Subprogram

A subroutine, unlike a function which always returns only one value, can return many values (or no values). Therefore, we use a subroutine when either several values are to be computed and returned or no values are to be returned (such as printing the values of some variables). The general structure of a subroutine is:

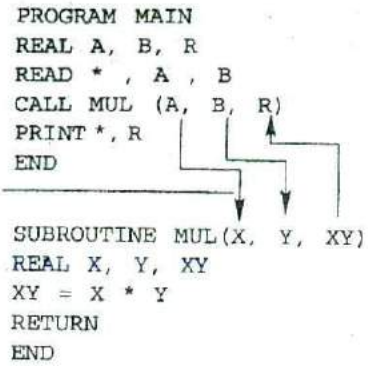
```
SUBROUTINE name (arguments)
Declaration of arguments
...
...          ←———————————  Execution statements
...
RETURN
END
```

where *name* is the subroutine name and *arguments* are *dummy* variables that must be declared for their type. When subroutine has no arguments, the parentheses are omitted (note that in case of function, parentheses are necessary even if there are no arguments). The outputs of subroutine are returned to the calling program by means of the arguments.

A subroutine can be invoked using the CALL statement as follows:

```
CALL name (arguments)
      or
CALL name
```

The actual arguments in the calling statement must agree in a one-to-one manner with the order and type of the arguments in the subroutine. Example:



```
PROGRAM MAIN
REAL A, B, R
READ * , A , B
CALL MUL (A, B, R)
PRINT *, R
END
```

```
SUBROUTINE MUL(X, Y, XY)
REAL X, Y, XY
XY = X * Y
RETURN
END
```

The calling program assigns the values of A and B to the variables X and Y in the subroutine which in turn assigns the value of XY (computed in the subroutine) to the variable R. Compare this with the function subprogram.

Note that the variables that are not passed as arguments may be passed to the subroutine using a COMMON statement.

FORTRAN 90 greatly extends the power of function subprograms by allowing the result to be an array or structure. Function subprograms are designed as follows:

```
FUNCTION name(arguments) RESULT (result -
                                  variable)
Declaration of arguments and result-variable
...
...
result-variable = expression
END FUNCTION name
```

Instead of function name, the result-variable is assigned the value that is to be returned to the calling function. The result-variable is a variable name that has been placed like a function argument with the RESULT keyword, immediately after the function name. Both the arguments and the result-variable are declared for their types.

Note that, in FORTRAN 90, all programs and subprograms use the name of the program or subprogram in the END statement as follows:

```
END FUNCTION F
END SUBROUTINE SWAP
END PROGRAM SORT
```

FORTRAN 90 also includes features such as optional arguments, keyword-identified arguments and array sizes which are very powerful compared to FORTRAN 77. These features must be used wherever possible.

FORTRAN allows us to write out a formula for a function and define it using the assignment statement inside the program itself (instead of using an "external" function subprogram). Since such functions are "one-line" functions, they are called *statement functions*. A statement function is defined as follows:

Function-name (*arguments*) = expression

where *expression* is the FORTRAN expression of the formula (or function) to be evaluated and *arguments* is a list of variables used in the expression. The arguments are simple integer or real variables. Examples:

```
AREA  (R)            = 3.1416 * R * R
VALUE  (P,  R)       = P  *  (1.0 + R) ** N
POLY  (X,  Y,  M,  N) = X ** M + Y ** N
```

A variable which appears in the expression but is not defined as an argument is called the *parameter* of the function. Values of such variables should be defined before using the function.

The function can be used in any subsequent lines of the program by writing the name of the function with actual arguments, like

```
CIRCLE  = AREA(X)
FVALUE  = VALUE(AMOUNT,  INTEREST)
POLY1   = POLY  (A,  2,  B,  2)
RING    = AREA(X1)  - AREA(X2)
```

Note that the functions can be used on the right side like any other variables. The actual arguments may be variables or constants (or even expressions). However, they must agree in number order and type with the dummy arguments in the function definition statement.

A statement function may use other statement functions if they are defined before it. Like function subprograms, the statement functions must be declared for their type in the program and defined after all declarations, but before the first executable statement.

## 5.10 INTRINSIC FUNCTIONS

In numerical computing, we use mathematical functions like logarithm, square root, absolute value, sine, etc., very frequently. FORTRAN supports a library of such functions which can be invoked in our programs. Since these functions are part of FORTRAN, they are also called *intrinsic* or *built-in functions*. An intrinsic function can be invoked by simply typing the name of the function followed by the arguments enclosed in parentheses. Example:

```
ABS(X)     COS(THETA)     SQRT(X * X + Y * Y)
```

The most commonly used intrinsic mathematical functions are summarised in Table 5.5. When using any of these functions, it is a good practice to declare them using the INTRINSIC statement in the declaration section.

**Table 5.5** Commonly used mathematical functions

| *Function* | *Description* |
|---|---|
| ABS(x) | Absolute value |
| ACOS(x) | Arccosine<br>(result in radians) |
| ASIN(x) | Arcsine<br>(result in radians) |
| ATAN(x) | Arctangent<br>(result in radians) |
| ATAN2($x_1$, $x_2$) | Arctangent of $x_1/x_2$<br>(result in radians) |
| COS(x) | Cosine<br>($x$ in radians) |
| COSH(x) | Hyperbolic cosine |
| DBLE(x) | Conversion to double precision real |
| EXP(x) | Power of $e$ |
| INT(x) | Truncation to integer |
| LOG(x) | Natural logarithm<br>(base $e$) |
| LOG10(x) | Common logarithm<br>(base 10) |
| MAX($x_1$, $x_2$, ...) | Maximum value |
| MIN($x_1$, $x_2$, ...) | Minimum value |
| MOD($x_1$, $x_2$) | Remainder of division $x_1$, $x_2$<br>(e.g. MOD(5,3) is 2) |
| NINT(x) | Conversion to nearest integer |
| REAL(x) | Conversion to single-precision real |
| SIN(x) | Sine<br>($x$ in radians) |
| SINH(x) | Hyperbolic sine |
| SQRT(x) | Square root<br>$x \geq 0.0$ |
| TAN(x) | Tangent<br>($x$ in radians) |
| TANH(x) | Hyperbolic tangent |

## 5.11 DEBUGGING, TESTING AND DOCUMENTATION

Errors in programming are common. Therefore, a program must be tested for any errors before it is used. Errors in computer code are called *bugs* and the process of correcting them is called *debugging*.

The program may be grammatically error free but may produce wrong results. Sometimes, a program may produce correct results for one set of data and wrong results for another set. Such errors are due to improper program logic and are therefore known as *logic errors* or *run-time errors*. It is a good practice to test the program for all possible range and combination of data.

Documentation is a most important but often neglected activity by the programmers. Documentation provides all details about the program intent, its variables and other requirements that allow the users to immediately understand and implement the program more easily.

Documentation includes two parts -- internal documentation and external documentation. Internal documentation means the use of explanatory remarks throughout the program, which describe how various parts of the program work. This is very important from the maintenance point of view.

External documentation includes instructions to the users on how to implement the program and what actions should be taken in certain special circumstances. Such a document is called *user manual*.

## 5.12 SUMMARY

We presented an overview of FORTRAN 77 in this Chapter. We discussed briefly the following features that are frequently used in developing numerical computing software:

- various categories of data types used to represent numbers and characters
- rules of defining variable names and creating storage space for them
- creation and use of subscripted variables to represent tables of data
- input/output statements required to read data values and print results
- operators used for evaluating mathematical and logical expressions
- control structures supported by FORTRAN 77
- design and use of subprograms in building a large application program

We have also highlighted the FORTRAN 90 features wherever applicable.

### Key Terms

*Algebraic expression*
*Arguments*
*Arithmetic operators*
*Array*
*Assignment statement*
*Backward jump*
*Bugs*
*Built-in functions*
*Calling program*
*Calling statement*

*Logic errors*
*Logical constants*
*Logical expression*
*Logical operators*
*Logical type*
*Looping structure*
*Main program*
*Mixed-mode expression*
*Modular programming*
*Modules*

(Contd.)

*(Contd.)*

*Character constants*
*Character type*
*Comment line*
*Complex constants*
*Complex type*
*Conditional execution*
*Constants*
*Control statement*
*Control variable*
*Counter*
*Debugging*
*Dimension*
*DO loop*
*DO..WHILE structure*
*Documentation*
*Double precision*
*Driver program*
*Dummy variables*
*Executable statement*
*Exponential form*
*Expressions*
*Flow chart*
*Format-directed I/O statement*
*FORTRAN expression*
*Forward jump*
*Functions*
*IF..ELSE structure*
*Initialisation*
*Inner loop*
*Input statement*
*Integer constants*
*Integer type*
*Intrinsic functions*
*List-directed I/O statement*
*Multidimensional array*
*Named constant*
*Nesting*
*Non-executable statement*
*One-dimensional array*
*Operators*
*Outer loop*
*Output statement*
*Parameter statement*
*Positional form*
*Precedence rule*
*Processing block*
*Program statement*
*Program unit header*
*Real constants*
*Real type*
*Relational operators*
*Run-time errors*
*Size*
*Statement functions*
*Statement label*
*Statement number*
*Subprogram*
*Subroutine*
*Subscripted variable*
*Subscripts*
*Symbolic constant*
*Testing*
*Two-dimensional array*
*Type declaration*
*User manual*
*Variable declaration*
*Variables*

## REVIEW QUESTIONS

1. What are FORTRAN constants?
2. What are logical constants? Where are they used?
3. When do we use the exponential form to represent real numbers?
4. What are variables? State the rules of naming variables?
5. What is an array? When do we use arrays in computing?
6. What is meant by declaration of variables? How are array variables declared in FORTRAN?
7. Describe the actions of the following statements:
   (a) READ *
   (b) PRINT *

(c) READ (*, 100)
(d) WRITE (*, 200)
(e) WRITE (6, 200)
(f) WRITE (*,*)

8. What is the function of a FORMAT statement? Give an example.
9. How are DATA statements used to provide values to variables?
10. State the hierarchy of operations followed by FORTRAN in evaluating expressions.
11. List FORTRAN statements that are used to implement conditional execution statements. How are they different in terms of implementation?
12. Give two examples of each of the following expressions:
    (a) Relational expression
    (b) Logical expression
    (c) Mixed-made expression

    Is there any special caution to be exercised in writing these expressions? Explain.
13. Why should we avoid the use of real variables as DO loop parameters?
14. What is nesting? When do we need to use nesting in numerical computing?
15. What are subprograms? How are they used in program development?
16. Distinguish between the function subprogram and subroutine subprogram.
17. What is a statement function? How is it different from function subprogram?
18. Give at least two examples of using statement functions in numerical computing?
19. What is testing? How is it different from debugging?
20. Describe the importance of documentation for programmers and program users.

## REVIEW EXERCISES

1. Which of the following are illegal FORTRAN names? Why?
    (a) TOTAL (b) PART - 1
    (c) % MARK (d) REAL
    (e) A+ (f) 3M
    (g) X23 (h) X AXIS
2. Classify each of the following constants as an integer constant or a real constant. If they are neither, state the reasons.
    (a) 123 (b) 123. (c) '123'
    (d) 12 + 3 (e) −12.3 (f) +12
    (g) 12/3 (h) 1,234 (i) FOUR
    (j) $6.5 (k) 0.12E3 (l) −1.5E02
    (m) E5 (n) 5E.5 (o) 25.3–

3. Which of the following are legal character constants?
   (a) ‘A’ (b) TOTAL’
   (c) ‘1.23’ (d) ‘TOTA MARKS’
   (e) ‘NEWTON’S LAW’ (f) ‘A.B.RAM’

4. Write the FORTRAN expressions for the following:

   (a) $\frac{a}{c+d} \times \frac{e}{f}$ (c) $\frac{a+b}{c} - b\,(d-e)\,(a \times \boldsymbol{d})^{2}$

   (b) $ax^2y + bxy^2 + c$ (d) $\left(c + \frac{x}{y}\right) n - 1$

5. Find the values of M and A when each of the following arithmetic statements is executed.
   (a) A = 2.5 + 3.0 ** 2/3.0
   (b) A = 2.0 ** 2 + 3.0 ** 2 – 4.0 * 3.0/2.0
   (c) A = 16/2 ** 3 + 5/2 * (2 * (6 – 4))
   (d) M = (9/4)/(3/2)
   (e) M = 4 ** 2 * (2/3)

6. Identify errors in the following assignment statements:
   (a) X = SUM/N
   (b) A = COS(X) + FLOAT(N)
   (c) N = (X/Y) * (X/Z)
   (d) M–1 = (A + B + C)/D
   (e) W = X ** –2 + SQRT(N)
   (f) D = P * ALOG (–3.5)

7. Following statements contain mixed mode expressions. Correct them using
   (a) the type declaration statements
   (b) the type conversion functions
   (c) none of the above
   (1) AREA = LENGTH * WIDTH
   (2) FORCE = MASS * ACCEL
   (3) DIST = SQRT(N ** 2)

8. Using library functions construct FORTRAN statements for the following:

   (a) $A = \sqrt{s(s-a)(s-b)(s-c)}$

   (b) $c = \sqrt{a^{2} + b^{2} - 2ab\cos(x)}$

   (c) $z = \sin\left|\frac{x-y}{x+y}\right|$

   (d) $f = \frac{1}{2\pi} \times e - \frac{(x+y)^{2}}{2}$

9. Given below are three sets of expressions. The two expressions in each set, though appear to be identical, do not produce the same results for certain values of integer variables *I*, *J* and *K* and the

real variable $X$. Identify those values for which the two expressions are not equal.

(a) $(I + J) / K$ and $I/K + J/K$

(b) $I * (J/K)$ and $I * J/K$

(c) $X * I/J$ and $X * (I/J)$

10. Write a program to read two values from the keyboard and to display their sum along with their values on the screen as follows:
    (a) All the three values in one line
    (b) Values one below the other in separate lines

11. Given the lengths of the two sides of a right triangle, write a program to do the following:
    (a) To read the values of two sides from the keyboard
    (b) To calculate the area of the triangle (one-half the product of two sides)
    (c) To calculate the length of hypotenuse (square root of the sums of squares of the sides) and
    (d) To print the results with labels like
    Area =
    Hypotenuse =

12. Write a program to evaluate the expression

$$w = \frac{(x + y)^2 - 2xy - y^2}{x^2}$$

and print the results for the following values of $x$ and $y$:

(a) $x = 0.05$ and $y = 900$

(b) $x = 0.005$ and $y = 900$

(c) $x = 0.002$ and $y = 900$

Are the results different?

Note that the above expression, on simplification, reduces to $w = 1$. If the results are different, why?

13. Declare the variables to be double precision in the above program and see the results. Is there a difference? If so, why?

14. Write a program which requests the user to type in a number. If the number is four digit long or longer, then the computer should provide a message that the number is too large. If it is two digit long or shorter, then the message should be that the number is too small. Otherwise, print a message

WELL DONE, WE THINK ALIKE

15. Write a program to solve the quadratic equation

$$ax^2 + bx + c = 0$$

by using the formula

$$\text{roots} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The program should display the real roots or message indicating that there are no real roots (if $b^2 - 4ac$ is negative).

16. Write a program to read the marks obtained by 25 students in a class and count the number of students with marks in the following range:
    (a) 0 to 39 (Fail)
    (b) 40 to 59 (Pass)
    (c) 60 and above (Pass with I class)
17. Write programs to evaluate the following functions to 0.0001 per cent accuracy.

    (a) $\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$

    (b) $\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots$
18. Write a program to read a set of numbers, count them, and find and print the largest and smallest numbers in the list and their positions in the list.
19. Write a program to calculate and print the mean, variance, and standard deviation of a set of $N$ numbers.

$$\text{Mean} = \frac{1}{N}\sum_{i=1}^{N} x_i$$

$$\text{Variance} = \frac{1}{N}\sum_{i=1}^{N} x_i^2 - \frac{1}{N^2}\left(\sum_{i=1}^{N} x_i\right)^2$$

$$\text{Standard Deviation} = \sqrt{\text{Variance}}$$

20. Write a program to find the largest element of a given matrix and print out the value with location details.
21. Write a subprogram to evaluate the factorial of a number which is given by

$$n\,! = n(n-1)\,(n-2)\;....\;1$$

Using this subprogram write a main program to calculate the binomial coefficient

$$b = \frac{n!}{(n-r)!\,r!}$$

This gives the number of combinations of $n$ objects take $r$ at a time.

22. Write a subroutine subprogram that will interchange the values of two variables when called.
23. Write a menu-driven program that allows the user to use one of the following options:
    (a) To convert miles to kilometres
    (b) To convert feet to metres
    (c) To convert degrees Fahrenheit to degree Celsius

**Note:** 1 mile = 1.60935 kilometres
1 foot = 0.3048 metres
C = 5/9 (F − 32)

24. Rewrite the following program so that it will take minimum time for execution.

```
      READ (5, 111) W, C1, C2, G, R, V
      CRT1 = W * C1 * (1.0 + G * R) * V
      CRT2 = W * (C2 + C1 * (1.0 + G * R)) * V
      CAP = C2 + C1 * (1.0 + G * R)
      WRITE (6, 222) CRT1, CRT2, CAP
  111 FORMAT (6F10.2)
  222 FORMAT (1Hb, 2F10.2, 5X, F10.5)
      STOP
      END
```

25. Improve the following program segments:

```
1.          READ (5, 11) X, Y
            DO 50 I = 1, 100
            AI = (X * X + Y * Y)/AI
        50  WRITE (6, 22) A
        11  FORMAT (2F5.2)
        22  FORMAT (1Hb, F10.5)
            STOP
            END
2.          . . .
            . . .
            DO 50 I = 1, 50
            . . .
            . . .
            X1 = X + Y/B * C
            X2 = Z + Y/B * C
            . . .
            . . .
            CONTINUE
            . . .
            . . .
```

CHAPTER 6

# Roots of Nonlinear Equations

## 6.1 INTRODUCTION

Mathematical models for a wide variety of problems in science and engineering can be formulated into equations of the form

$$f(x) = 0 \tag{6.1}$$

where $x$ and $f(x)$ may be real, complex, or vector quantities. The solution process often involves finding the values of $x$ that would satisfy the Eq. (6.1). These values are called the *roots* of the equation. Since the function $f(x)$ becomes zero at these values, they are also known as the zeros of the function $f(x)$.

Equation (6.1) may belong to one of the following types of equations:

1. Algebraic equations
2. Polynomial equations
3. Transcendental equations

(Any function of one variable which does not graph as a straight line in two dimensions, or any function of two variables which does not graph as a plane in three dimensions, can be said to be *nonlinear*.) Consider the function

$$y = f(x)$$

$f(x)$ is a *linear* function, if the dependent variable $y$ changes in direct proportion to the change in independent variable $x$. For example

$$\boxed{y = 3x + 5}$$

is a *linear* function.

On the other hand, $f(x)$ is said to be nonlinear, if the response of the dependent variable $y$ is not in direct or exact proportion to the changes in the independent variable $x$. For example

$$y = x^2 + 1$$

is a nonlinear function.

There are many situations in science and engineering where the relationship between variables is *nonlinear*.

## Algebraic Equations

An equation of type $y = f(x)$ is said to be *algebraic* if it can be expressed in the form

$$f_n\, y_n + f_{n-1}\, y_{n-1} + \dots + f_1\, y_1 + f_0 = 0 \tag{6.2}$$

where $f_i$ is an $i$th order polynomial in $x$. Equation (6.2) can be thought of as having a general form

$$f(x, y) = 0 \tag{6.3}$$

This implies that Eq. (6.3) portrays a dependence between the variables $x$ and $y$. Some examples are:

1. $3x + 5y - 21 = 0$ (linear)
2. $2x + 3xy - 25 = 0$ (non-linear)
3. $x^3 - xy - 3y^3 = 0$ (non-linear)

These equations have an infinite number of pairs of values of $x$ and $y$ which satisfy them.

## Polynomial Equations

Polynomial equations are a simple class of algebraic equations that are represented as follows:

$$a_n\, x^n + a_{n-1}\, x^{n-1} + \dots + a_1\, x + a_0 = 0 \tag{6.4}$$

This is called $n^{\text{th}}$ degree polynomial and has $n$ roots. The roots may be

1. real and different
2. real and repeated
3. complex numbers

Since complex roots appear in pairs, if $n$ is odd, then the polynomial has at least one real root. For example, a cubic equation of the type

$$a_3x^3 + a_2x^2 + a_1x + a_0 = 0$$

will have at least one real root and the remaining two may be real or complex roots. Some specific examples of polynomial equations are:

1. $5x^5 - x^3 + 3x^2 = 0$
2. $x^3 - 4x^2 + x + 6 = 0$
3. $x^2 - 4x + 4 = 0$

## Transcendental Equations

A non-algebraic equation is called a *transcendental equation*. These include trigonometric, exponential and logarithmic functions. Examples of transcendental equation are:

1. $2 \sin x - x = 0$
2. $e^x \sin x - 1/2\, x = 0$
3. $\log x^2 - 1 = 0$
4. $x - e^{1/x} = 0$

A transcendental equation may have a finite or an infinite number of real roots or may not have real root at all.

## 6.2 METHODS OF SOLUTION

There are a number of ways to find the roots of nonlinear equations such as those described in Section 6.1. They include:

1. Direct analytical methods
2. Graphical methods
3. Trial and error methods
4. Iterative methods

In certain cases, roots can be found by using *direct analytical methods*. For example, consider a quadratic equation such as

$$ax^2 + bx + c = 0 \tag{6.5}$$

We know that the solution of this equation is

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{6.6}$$

Equation (6.6) gives the two roots of equation (6.5). However, there are equations that cannot be solved by analytical methods. For example, the simple transcendental equation

$$2 \sin x - x = 0$$

cannot be solved analytically. Direct methods for solving non-linear equations do not exist except for certain simple cases.

*Graphical methods* are useful when we are satisfied with approximate solution for a problem. This method involves plotting the given function and determining the points where it crosses the $x$-axis. These points represent approximate values of the roots of the function.

Another approach to obtain approximate solution is the trial and error technique. This method involves a series of guesses for $x$, each time evaluating the function to see whether it is close to zero. The value of $x$ that causes the function value closer to zero is one of the approximate roots of the equation.

Although graphical and trial and error methods provide satisfactory approximations for many problem situations, they become cumbersome and time consuming. Moreover, the accuracy of the results are inadequate for the requirements of many engineering and scientific problems. With the advent of computers, algorithmic approaches known as *iterative methods* have become popular. An iterative technique usually begins with an approximate value of the root, known as the *initial guess*, which is then successively corrected iteration by iteration. The process of iteration stops when the desired level of accuracy is obtained. Since iterative methods involve a large number of iterations and arithmetic operations to reach a solution, the use of computers has become inevitable to make the task simple and efficient.

In this chapter, we shall discuss a few iterative methods of solution that are commonly used. These methods are designed to determine the value of a single real root using some initial guess values. Later in the chapter, we shall also discuss methods to determine all the roots of a polynomial. Finally, we shall discuss the solution of a system of non-linear equations.

## 6.3 ITERATIVE METHODS

There are a number of iterative methods that have been tried and used successfully in various problem situations. All these methods typically generate a sequence of estimates of the solution which is expected to converge to the true solution. As mentioned earlier, all iterative methods begin their process of solution with one or more guesses at the solution being sought. Iterative methods, based on the number of guesses they use, can be grouped into two categories:

1. Bracketing methods
2. Open end methods

*Bracketing methods* (also known as *interpolation* methods) start with two initial guesses that 'bracket' the root and then systematically reduce the width of the bracket until the solution is reached. Two popular methods under this category are:

1. Bisection method
2. False position method

These methods are based on the assumption that the function changes sign in the vicinity of a root.

*Open end methods* (also known as *extrapolation* methods) use a single starting value or two values that do not necessarily bracket the root. The following iterative methods fall under this category:

1. Newton-Raphson method
2. Secant method
3. Muller's method
4. Fixed-point method
5. Bairstow's method

It may be noted that the bracketing methods require to find sign changes in the function during every iteration. Open end methods do not require this.

## 6.4 STARTING AND STOPPING AN ITERATIVE PROCESS

### Starting the Process

Before an iterative process is initiated, we have to determine either an approximate value of root or a "search" interval that contains a **root.** One simple method of guessing starting points is to plot the curve of $f(x)$ and to identify a search interval near the root of interest. Graphical representation of a function cannot only provide us rough estimates of the roots, but also help us in understanding the properties of the function, thereby identifying possible problems in numerical computing. A plot of

$$f(x) = x^3 - x - 1$$

is shown in Fig. 6.1. Although $f(x)$ is a cubic function, it intersects the $x$-axis at only one point. This suggests that the remaining two roots are imaginary ones.

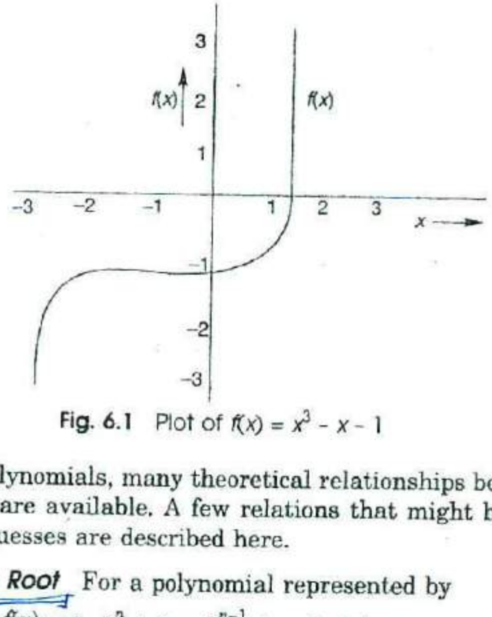


Fig. 6.1 Plot of $f(x) = x^3 - x - 1$

In the case of polynomials, many theoretical relationships between roots and coefficients are available. A few relations that might be useful for making initial guesses are described here.

***Largest Possible Root*** For a polynomial represented by

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots a_1 x + a_0 \tag{6.7}$$

the largest possible root is given by

$$x_1^* = -\frac{a_{n-1}}{a_n} \tag{6.8}$$

This value is taken as the initial approximation when no other value is suggested by the knowledge of the problem at hand.

***Search Bracket*** Another relationship that might be useful for determining the search intervals that contain the real roots of a polynomial is

$$|x^*| \leq \sqrt{\left(\frac{a_{n-1}}{a_n}\right)^2 - 2\left(\frac{a_{n-2}}{a_n}\right)} \tag{6.9}$$

where $x$ is the root of the polynomial. Then, the maximum absolute value of the root is

$$|x^*_{\max}| = \sqrt{\left(\frac{a_{n-1}}{a_n}\right)^2 - 2\left(\frac{a_{n-2}}{a_n}\right)^{3} - 3\left(\frac{a_{n-3}}{a_n}\right)} \tag{6.10}$$

This means that no root exceeds $x_{\max}$ in absolute magnitude and thus, all real roots lie within the interval $\left(-|x^*_{\max}|, |x^*_{\max}|\right)$.

There is yet another relationship that suggests an interval for roots. All real roots $x$ satisfy the inequality

$$|x^*| \leq 1 + \frac{1}{|a_n|} \max\{|a_0|, |a_1|, \ldots, |a_{n-1}|\} \tag{6.11}$$

where the "max" denotes the maximum of the absolute values $|a_0|$, $|a_1|, \ldots, |a_{n-1}|$.

**Example 6.1**

Consider the polynomial equation

$$\underset{a_n}{2}x^3 - \underset{a_{n-1}}{8}x^2 + \underset{a_{n-2}}{2}x + 12 = 0$$

Estimate the possible initial guess values.

The largest possible root is

$$x_1^* = -\frac{-8}{2} = 4$$

That is, no root can be larger than the value 4.
All roots must satisfy the relation

$$|x^*| \leq \sqrt{\left(\frac{-8}{2}\right)^2 - 2\left(\frac{2}{2}\right)} = \sqrt{14}$$

Therefore, all real roots lie in the interval $(-\sqrt{14}, \sqrt{14})$. We can use these two points as initial guesses for the bracketing methods and one of them for the open end methods.

### Stopping Criterion

An iterative process must be terminated at some stage. When ? We must have an objective criterion for deciding when to stop the process. We may use one (or combination) of the following tests, depending on the behaviour of the function, to terminate the process:

1. $|x_{i+1} - x_i| \quad \le E_a$ (absolute error in $x$)
2. $\left|\dfrac{x_{i+1} - x_i}{x_{i+1}}\right| \quad \le E_r$ (relative error in $x$), $x \ne 0$
3. $|f(x_{i+1})| \quad \le E$ (value of function at root)
4. $|f(x_{i+1}) - f(x_i)| \quad \le E$ (difference in function values)
5. $|f(x)| \quad \le F_{\max}$ (large function value)
6. $|x_i| \quad \le XL$ (large value of $x$)

Here, $x_i$ represents the estimate of the root at $i$th iteration and $f(x_i)$ is the value of the function at $x_i$.

There may be situations where these tests may fail when used alone. Sometimes even a combination of two tests may fail. A practical convergence test should use a combination of these tests. In cases where we do not know whether the process converges or not, we must have a limit on the number of iterations, like

Iterations $\ge N$ (limit on iterations).

## 6.5 EVALUATION OF POLYNOMIALS

All iterative methods require the evaluation of functions for which solution is sought. Since it is a recurring task, the design of an efficient algorithm for evaluating the function assumes a greater importance. While it is not possible to propose a general algorithm for evaluating transcendental functions, it is quite simple to design an algorithm for evaluating polynomials.

The polynomial is a sum of $n$+1 terms and can be expressed as

$$f(x) = \sum_{i=0}^{n} a_i\, x^i = a_0 + \sum_{i=1}^{n} a_i\, x^i \qquad (6.12)$$

This can be easily implemented using a DO loop in FORTRAN. This would require $n(n + 1)/2$ multiplications and $n$ additions.

**Example 6.2**

Write a FORTRAN program segment to implement Eq. (6.12).

```
         .  .  .

         .  .  .

         SUM  =  AO
                DO 100 I = 1, N
                     SUM = SUM + A(I) * X ** I
      100 CONTINUE
          .  .  .

          .  .  .

          .  .  .
```

Let us consider the evaluation of a polynomial using *Horner's Rule* as follows:

$$f(x) = ((\ldots((a_n\, x + a_{n-1})x + a_{n-2})x + \ldots + a_1)x + a_0) \qquad (6.13)$$

Here, the innermost expression $a_n x + a_{n-1}$ is evaluated first. The resulting value constitutes a multiplicand for the expression at the next level. The number of level equals $n$, the degree of polynomial. Note that this approach needs a total of $n$ additions and $n$ multiplications.

Horner's rule, also known as *nested multiplication*, is implemented using Algorithm 6.1. The quantities $p_n, p_{n-1}, \ldots, p_0$ are evaluated recursively. The final quantity $p_0$ gives the value of the function $f(x)$.

**Horner's Rule**

$$p_n = a_n$$

$$p_{n-1} = p_n x + a_{n-1}$$

$$\ldots$$

$$\ldots$$

$$p_j = p_{j+1}\, x + a_j$$

$$\ldots$$

$$\ldots$$

$$p_1 = p_2 x + a_1$$

$$f(x) = p_0 = p_1 x + a_0$$

**Algorithm 6.1**

### Example 6.3

Evaluate the polynomial

$$f(x) = x^3 - 4x^2 + x + 6$$

using Horner's rule at $x = 2$.

$n = 3$, $a_3 = 1$, $a_2 = -4$, $a_1 = 1$, and $a_0 = 6$

$$p_3 = a_3 = 1$$
$$p_2 = 1 \times 2 + (-4) = -2$$
$$p_1 = (-2) \times 2 + 1 = -3$$
$$p_0 = (-3) \times 2 + 6 = 0$$
$$f(2) = 0$$

## Program POLY

Program POLY shows a FORTRAN program to evaluate a polynomial of degree $n$ using Horner's rule. This program uses a subroutine HORNER to implement Horner's algorithm.

It is an interactive program and, therefore, requests input for degree of polynomial ($n$), polynomial coefficients ($a_i$) and value of $x$ from the user at the time of execution. Output of a sample run of the program POLY is given at the end of the program:

```
* --------------------------------------------------------- *
  PROGRAM POLY
* --------------------------------------------------------- *
* Main program                                              *
*    Program POLY evaluates a polynomial of degree n at     *
*    any point X using Horner's rule                        *
* --------------------------------------------------------- *
* Functions invoked                                         *
*    NIL                                                    *
* --------------------------------------------------------- *
* Subroutines used                                          *
*    HORNER                                                 *
* --------------------------------------------------------- *
* Variables used                                            *
*       N - Degree of polynomial                            *
*       A - Array of polynomial coefficients                *
*       X - Point of evaluation                             *
*       P - Value of polynomial at X                        *
* --------------------------------------------------------- *
* Constants used                                            *
*       NIL                                                 *
* --------------------------------------------------------- *
      INTEGER N
      REAL A, X, P
      EXTERNAL HORNER
      DIMENSION A(10)
      WRITE(*,*) 'Input degree of polynomial, N'
      READ(*,*) N
      WRITE(*,*) 'Input polynomial coefficients ( A(0)
                                                to A(N))'
      DO 100 I = 1, N+1
        READ(*,*) A(I)
100   CONTINUE
      WRITE(*,*) 'Input value of X (point of evaluation)'
      READ(*,*) X
* Evaluating polynomial at X using Horner's method
      CALL HORNER( N,A,X,P )
* Writing the result
      WRITE(*,*)
      WRITE(*,*) 'F(X) = ', P, ' at X = ', X
      WRITE(*,*)
      STOP
      END
* --------------- End of main program POLY --------- *
* -------------------------------------------------- *
      SUBROUTINE HORNER( N,A,X,P )
* -------------------------------------------------- *
* Subroutine                                         *
*   HORNER computes the value of a polynomial of order *
*   n at any given point x.                          *
* -------------------------------------------------- *
* Arguments                                          *
* Input                                              *
*     N - Degree of polynomial                       *
*     A - Polynomial coefficients (array of size N+1) *
*     X - Point of interest of evaluation            *
* Output                                             *
*     P - Value of polynomial at X                   *
* -------------------------------------------------- *
* Local Variables                                    *
*   NIL                                              *
* -------------------------------------------------- *
* Functions invoked                                  *
*   NIL                                              *
* -------------------------------------------------- *
* Subroutines called                                 *
*   NIL                                              *
* -------------------------------------------------- *
```

```
      REAL A, X, P
      INTEGER N
      DIMENSION A(10)
      P = A(N+1)
      DO 111 I = N, 1, -1
        P = P*X + A(I)
111   CONTINUE
      RETURN
      END
* ----------- End of Subroutine HORNER ------------- *
```

***Test Run Results***

```
      Input degree of polynomial, N
      3
      Input polynomial coefficients (A(0) to A(N))
      12
      5
      6
      2
      Input value of X (point of evaluation)
      0.125
      F(X) = 12.7226600 at X = 1.250000E - 001
```

The polynomial used for evaluation is

$$2x^3 + 6x^2 + 5x + 12$$

and the coefficients are represented by `A(1)`, `A(2)`, `A(3)`, and `A(4)` instead of `A(0)`, `A(1)`, `A(2)`, and `A(3)` in the program.

## 6.6 BISECTION METHOD

The *bisection method* is one of the simplest and most reliable of iterative methods for the solution of nonlinear equations. This method, also known as *binary chopping* or *half-interval* method, relies on the fact that if $f(x)$ is real and continuous in the interval $a < x < b$, and $f(a)$ and $f(b)$ are of opposite signs, that is,

$$f(a)\,f(b) < 0$$

then there is at least one real root in the interval between $a$ and $b$. (There may be more than one root in the interval).

Let $x_1 = a$ and $x_2 = b$. Let us also define another point $x_0$ to be the midpoint between $a$ and $b$. That is,

$$x_0 = \frac{x_1 + x_2}{2} \tag{6.14}$$

Now, there exists the following three conditions:

1. if $f(x_0) = 0$, we have a root at $x_0$.

2. if $f(x_0)\, f(x_1) < 0$, there is a root between $x_0$ and $x_1$. , $x_2 =$
3. if $f(x_0) f(x_2) < 0$, there is a root between $x_0$ and $x_2$.

It follows that by testing the sign of the function at midpoint, we can deduce which part of the interval contains the root. This is illustrated in Fig. 6.2. It shows that, since $f(x_0)$ and $f(x_2)$ are of opposite sign, a root lies between $x_0$ and $x_2$. We can further divide this subinterval into two halves to locate a new subinterval containing the root. This process can be repeated until the interval containing the root is as small as we desire.

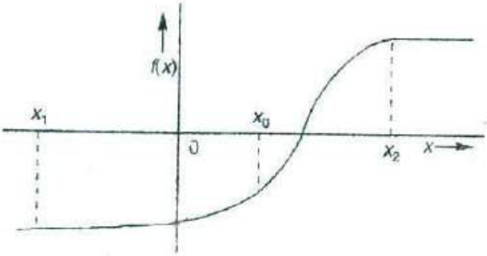


**Fig. 6.2** Illustration of bisection method

### Example 6.4 ✓

Find a root of the equation

$$x^2 - 4x - 10 = 0 \quad / \; 1 + 4 - 10 = -5$$

using bisection method.

The first step is to guess two initial values that would bracket a root. Using Eq. (6.10), we can decide the maximum absolute of the solution. Thus

$$x_{\max} = \sqrt{\left(\frac{-4}{1}\right)^2 - 2\left(\frac{-10}{1}\right)} = 6$$

Therefore, we have both the roots in the interval (–6, 6). The table below gives the values of $f(x)$ between –6 and 6 and shows that there is a root in the interval (–2, –1) and another in (5, 6).

| $x$ | –6 | –5 | –4 | –3 | –2 | –1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $f(x)$ | 50 | 35 | 22 | 11 | 2 | –5 | –10 | –13 | –14 | –13 | –10 | –5 | 2 |

Let us take $x_1 = -2$ and $x_2 = -1$.
Then

$$x_0 = \frac{-2-1}{2} = -1.5$$

$$f(-2) = 2 \text{ and } f(-1.5) = -1.75$$

$\nearrow f(x_1) f(x_0) \geq 0$

Since $f(-2)\, f(-1.5) < 0$, the root must be in the interval (–2, –1.5). The next step begins.

$$x_1 = -2,\ x_2 = -1.5 \text{ and } x_0 = -1.75$$
$$f(-1.75) = 0.0625$$

Since $f(-1.75)$ and $f(-1.5)$ are of opposite sign, the root lies in the interval $(-1.75, -1.5)$. Another iteration begins.

$$x_1 = -1.75,\ x_2 = -1.5 \text{ and } x_0 = -1.625$$
$$f(-1.625) = -0.859$$

Now, the root lies in the interval $(-1.75, -1.625)$

$$x_0 = -1.6875$$
$$f(-1.6875) = -0.40$$

Next $$x_0 = -\frac{1.75 + 1.6875}{2} = -1.72$$
$$f(-1.72) = -0.1616$$

Next $$x_0 = -\frac{1.75 + 1.72}{2} = -1.735$$
$$f(-1.735) = -0.05$$

Next $$x_0 = -1.7425$$
$$f(-1.7425) = +0.0063$$

The root lies between $-1.735$ and $-1.7425$.
Approximate root is $-1.7416$.

An algorithm to achieve this is given in Algorithm 6.2.

### Bisection Method

```
1. Decide initial values for $x_1$ and $x_2$ and stopping criterion, $E$.
2. Compute $f_1 = f(x_1)$ and $f_2 = f(x_2)$.
3. If $f_1 \times f_2 > 0$, $x_1$ and $x_2$ do not bracket any root and go to step 7;
   Otherwise continue.
4. Compute $x_0 = (x_1 + x_2)/2$ and compute $f_0 = f(x_0)$
5. If $f_1 \times f_0 < 0$ then
         set $x_2 = x_0$
   else
         set $x_1 = x_0$
         set $f_1 = f_0$
6. If absolute value of $(x_2 - x_1)/x_2$ is less than error $E$, then
         root = $(x_1 + x_2)/2$
         write the value of root
         go to step 7
   else
         go to step 4
7. Stop.
```

Algorithm 6.2

## Convergence of Bisection Method

In the bisection method, we choose a midpoint $x_0$ in the interval between $x_1$ and $x_2$. Depending on the sign of functions $f(x_0)$, $f(x_1)$, and $f(x_2)$, $x_1$ or $x_2$ is set equal to $x_0$ such that the new interval contains the root. In either case, the interval containing the root is reduced by a factor of 2. The same procedure is repeated for the new interval. If the procedure is repeated $n$ times, then the interval containing the root is reduced to the size

$$\frac{x_2 - x_1}{2^n} = \frac{\Delta x}{2^n}$$

After $n$ iterations, the root must lie within $\pm \Delta x/2^n$ of our estimate. This means that the error bound at $n^{\text{th}}$ iteration is

$$E_n = \left|\frac{\Delta x}{2^n}\right|$$

Similarly,

$$E_{n+1} = \left|\frac{\Delta x}{2^{n+1}}\right| = \frac{E_n}{2} \tag{6.15}$$

That is, the error decreases linearly with each step by a factor of 0.5. The bisection method is, therefore, *linearly convergent*. Since the convergence is slow to achieve a high degree of accuracy, a large number of iterations may be needed. However, the bisection algorithm is guaranteed to converge.

## Program BISECT

P

This program finds a root of a nonlinear equation using the bisection method. BISECT uses a subroutine, BIM to find a root in a given interval and invokes a function subprogram, F(x) to evaluate the function at the estimated root.

The subroutine subprogram BIM locates a root in the given interval [A, B] using Algorithm 6.2. BIM applies the following criterion for terminating the process

$$\left|\frac{x_n - x_{n-1}}{x_n}\right| < \text{EPS}$$

That is, the relative error in the successive approximations must be less than a specified error limit.

The function subprogram $F(x)$ simply evaluates the function value at a given value of $x$ and returns the result to the calling module. Note that by simply changing the function definition statement

$$F = x * x + x - 2$$

we can use the BISECT program to evaluate a root of any function.

**Also note that the program prints out a message in case the specified interval does not bracket a root.**

```
* -------------------------------------------------------- *
  PROGRAM BISECT
* -------------------------------------------------------- *
* Main program                                              *
*   This program finds a root of a nonlinear equation      *
*   using the bisection method                              *
* -------------------------------------------------------- *
* Functions invoked                                         *
*   F                                                       *
* -------------------------------------------------------- *
* Subroutines used                                          *
*   BIM                                                     *
* -------------------------------------------------------- *
* Variables used                                            *
*   A - Left endpoint of interval                           *
*   B - Right endpoint of interval                          *
*   S - Status                                              *
*   ROOT - Final Solution                                   *
*   COUNT - Number of Iterations done                       *
* -------------------------------------------------------- *
* Constants used                                            *
*   EPS - Error bound                                       *
* -------------------------------------------------------- *
    REAL A,B,ROOT,EPS,F
    INTEGER S, COUNT
    EXTERNAL BIM,F
    PARAMETER(EPS=0.000001)
    WRITE(*,*)
    WRITE(*,*) ' SOLUTION BY BISECTION METHOD''
    WRITE(*,*)
    WRITE(*,*) 'Input starting values'
    READ(*,*) A,B
    CALL BIM(A,B,EPS,S,ROOT,COUNT)
    IF (S.EQ.0) THEN
      WRITE(*,*)
      WRITE(*,*) 'Starting points do not bracket any root'
      WRITE(*,*) '(Check whether they bracket EVEN roots)'
      WRITE(*,*)
    ELSE
      WRITE(*,*)
      WRITE(*,*) 'Root = ', ROOT
      WRITE(*,*) 'F(Root)   = ', F(ROOT)
      WRITE(*,*)
      WRITE(*,*) 'ITERATIONS = ', COUNT
      WRITE(*,*)
      ENDIF
      STOP
      END
* -------------- End of main program ---------------- *
* -------------------------------------------------- *
      SUBROUTINE BIM(A,B,EPS,S,ROOT,COUNT)
* -------------------------------------------------- *
* Subroutine                                         *
*   This subroutine finds a root of nonlinear equation  *
*   in the interval [A,B] using the bisection method *
* -------------------------------------------------- *
* Arguments                                          *
* Input                                              *
*   A - Left endpoint of interval                    *
*   B - Right endpoint of interval                   *
*   EPS - Error bound                                *
* Output                                             *
*   S - Status                                       *
*   ROOT - Final Solution                            *
*   COUNT - Number of Iterations                     *
* -------------------------------------------------- *
* Local Variables                                    *
*   X1,X2,X0,F0,F1,F2                                *
* -------------------------------------------------- *
* Functions invoked                                  *
*   F,ABS                                            *
* -------------------------------------------------- *
* Subroutines called                                 *
*   NIL                                              *
* -------------------------------------------------- *
      REAL A,B,ROOT,EPS,F,X1,X2,X0,F0,F1,F2,ABS
      INTEGER S,COUNT
      EXTERNAL F
      INTRINSIC ABS
* Function values at initial points
      X1 = A
      X2 = B
      F1 = F(A)
      F2 = F(B)
* Test if initial values bracket a SINGLE root
      IF(F1*F2 .GT.0) THEN
         S = 0
         RETURN
      ENDIF
* Bisect the interval and locate the root iteratively
      COUNT = 1
```

```
111 X0 = (X1+X2)/2.0
    F0 = F(X0)
    IF (F0 .EQ. 0) THEN
      S=1
      ROOT = X0
      RETURN
    ENDIF
    IF(F1*F0 .LT.0) THEN
      X2 = X0
    ELSE
      X1 = X0
      F1 = F0
    ENDIF
* Test for accuracy and repeat the process, if necessary
    IF(ABS((X2-X1)/X2).LT.EPS) THEN
      S = 1
      ROOT = (X1+X2)/2.0
      RETURN
    ELSE
      COUNT = COUNT + 1
      GO TO 111
    ENDIF
    END
* ------------- End of subroutine BIM --------------- *
* ------------------------------------------------------ *
* Function subprogram F(x)
* ------------------------------------------------------ *
    REAL FUNCTION F(X)
    REAL X
    F = X*X+X-2
    RETURN
    END
* ------------- End of function F(X) ---------------- *
```

***Test Results of BISECT***

The program was used to solve the equation

$$x^2 + x - 2 = 0$$

using two sets of starting points:

(0.0, 2.0) and (0.5, 2.0)

*First run*

```
        SOLUTION BY BISECTION METHOD
      Input starting values
      0.0 2.0
```

```
Root  =  1.0000000
F(ROOT)  =  .0000000
ITERATIONS  =  1
Stop - Program terminated.
```

*Second run*

```
    SOLUTION BY BISECTION METHOD
Input starting values
0.5 2.0
Root           =     9.999999E-001
F(ROOT)        =    -3.576279E-007
ITERATIONS     =     21
Stop - Program terminated.
```

## 6.7 FALSE POSITION METHOD

In bisection method, the interval between $x_1$ and $x_2$ is divided into two equal halves, irrespective of location of the root. It may be possible that the root is closer to one end than the other as shown in Fig. 6.3. Note that the root is closer to $x_1$. Let us join the points $x_1$ and $x_2$ by a straight line. The point of intersection of this line with the $x$ axis ($x_0$) gives an improved estimate of the root and is called the *false position* of the root. This point then replaces one of the initial guesses that has a function value of the same sign as $f(x_0)$. The process is repeated with the new values of $x_1$ and $x_2$. Since this method uses the false position of the root repeatedly, it is called the *false position method* (or *regula falsi* in Latin). It is also called the *linear interpolation method* (because an approximate root is determined by linear interpolation).

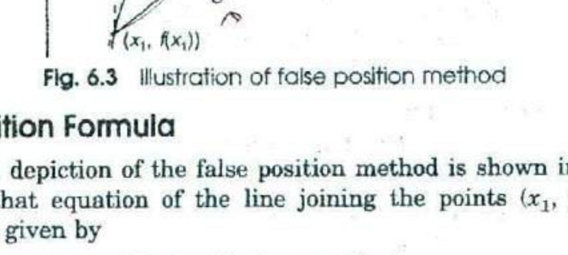


**Fig. 6.3** Illustration of false position method

### False Position Formula

A graphical depiction of the false position method is shown in Fig. 6.3. We know that equation of the line joining the points $(x_1, f(x_1))$ and $(x_2, f(x_2))$ is given by

$$\frac{f(x_2)-f(x_1)}{x_2-x_1}=\frac{y-f(x_1)}{x-x_1} \tag{6.16}$$

Since the line intersects the $x$-axis at $x_0$, when $x = x_0$, $y = 0$, we have

$$\frac{f(x_2)-f(x_1)}{x_2-x_1}=\frac{-f(x_1)}{x_0-x_1}$$

or

$$x_0-x_1=-\frac{f(x_1)(x_2-x_1)}{f(x_2)-f(x_1)}$$

Then, we have

$$\boxed{x_0=x_1-\frac{f(x_1)(x_2-x_1)}{f(x_2)-f(x_1)}} \tag{6.17}$$

This equation is known as the *false position formula*. Note that $x_0$ is obtained by applying a correction to $x_1$.

## False Position Algorithm

Having calculated the first approximate to the root, the process is repeated for the new interval, as done in the bisection method, using Algorithm 6.3.

**False Position Method**

```
Let $x_0 = x_1 - f(x_1) \times \frac{x_2 - x_1}{f(x_2) - f(x_1)}$
If $f(x_0) \times f(x_1) < 0$
        set $x_2 = x_0$
otherwise
        set $x_1 = x_0$
```

**Algorithm 6.3**

A major difference between this algorithm and the bisection algorithm is the way $x_0$ is computed.

**Example 6.5**

Use the false position method to find a root of the function

$$f(x) = x^2 - x - 2 = 0$$

in the range $1 < x < 3$

*Iteration 1*

Given $x_1 = 1$ and $x_2 = 3$

$$f(x_1) = f(1) = -2$$
$$f(x_2) = f(3) = \phantom{-}4$$

$$x_0 = x_1 - f(x_1) \times \frac{x_2 - x_1}{f(x_2) - f(x_1)}$$

$$= 1 + 2 \times \frac{3-1}{4+2} = 1.6667$$

*Iteration 2*

$$f(x_0)\,f(x_1) = f(1.6667)f(1) = 1.7778$$

Therefore, the root lies in the interval between $x_0$ and $x_2$. Then,

$$x_1 = x_0 = 1.6667$$
$$f(x_1) = f(1.6667) = -0.8889$$
$$f(x_2) = f(3) = 4$$

$$x_0 = 1.6667 + 0.8889 \times \frac{3 - 1.6667}{4 + 0.8889} = 1.909$$

*Iteration 3*

$$f(1.909)\,f(1.6667) = +0.2345$$

Root lies between $x_0(= 1.909)$ and $x_2(=3)$

Therefore,

$$x_1 = x_0 = 1.909$$
$$x_2 = 3$$

$$x_0 = 1.909 + 0.2647 \times \boxed{\frac{3 - 1.909}{4 - 0.2647}}$$

$$= 1.909 + 0.2647 \times \frac{1.091}{3.7353} = 1.986$$

The estimated root after third iteration is 1.986. Remember that the interval contains a root $x = 2$. We can perform additional iterations to refine this estimate further.

## Convergence of False Position Method

The false position formula is based on the linear interpolation model. In the false position iteration, one of the starting points is fixed while the other moves towards the solution. Assume that the initial points bracketing the solution are $a$ and $b$ and that $a$ moves towards the solution and $b$ is fixed as illustrated in Fig. 6.4.

Let $x_1 = a$ and $x_r$ be the solution.

Then,

$$e_1 = x_r - x_1$$
$$e_2 = x_r - x_2$$

That is,

$$e_i = x_r - x_i$$

It can be shown that

$$e_{i+1} = e_r \times \frac{(x_r - b)\,f''(R)}{f'(R)} \qquad (6.18)$$

where $R$ is some point in the interval $x_i$ and $b$. This shows that the process of iteration *converges linearly*.

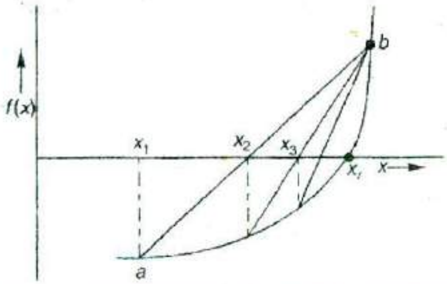


**Fig. 6.4** Convergence of false position method

## Program FALSE

The program FALSE finds a root of a nonlinear equation using the false position method. The program uses a function subprogram F and a subroutine, FAL to implement the method.

The function evaluates the function at any given point and the subroutine determines a root in a given interval using Algorithm 6.3.

We can use the FALSE program to identify a root of any function by changing the function statement in the function subprogram F.

```
* ------------------------------------------------------------ *
      PROGRAM FALSE
* ------------------------------------------------------------ *
* Main program                                                 *
*     This program finds a root of a nonlinear equation        *
*     by false position method                                 *
* ------------------------------------------------------------ *
* Functions invoked                                            *
*     F                                                        *
* ------------------------------------------------------------ *
* Subroutines used                                             *
*     FAL                                                      *
* ------------------------------------------------------------ *
* Variables used                                               *
*     A - Left endpoint of interval                            *
*     B - Right endpoint of interval                           *
*     S - Status                                               *
*     ROOT - Final solution                                    *
*     COUNT - Number of iterations completed                   *
* ------------------------------------------------------------ *
* Constants used                                               *
*     EPS - Error bound                                        *
* ------------------------------------------------------------ *
      REAL A,B,ROOT,EPS,F
      INTEGER S,COUNT
      EXTERNAL FAL,F
      PARAMETER(EPS = 0.000001)
      WRITE(*,*) 'Input starting values'
      READ(*,*) A,B
      WRITE(*,*)
      WRITE(*,*) ' SOLUTION BY FALSE POSITION METHOD'
      WRITE(*,*)
      CALL FAL(A,B,EPS,S,ROOT,COUNT)
      IF(S.EQ.0) THEN
        WRITE(*,*) 'Starting points do not bracket any
                                                   root'
        WRITE(*,*)
      ELSE
        WRITE(*,*)
        WRITE(*,*) 'Root = ', ROOT
        WRITE(*,*) 'F(ROOT) = ', F(ROOT)
        WRITE(*,*) 'NO.OF ITERATIONS = ', COUNT
        WRITE(*,*)
      ENDIF
      STOP
      END
* -------------- End of main FALSE ----------------- *
* ------------------------------------------------- *
      SUBROUTINE FAL(A,B,EPS,S,ROOT,COUNT)
* ------------------------------------------------- *
* Subroutine                                          *
*   FAL finds a root of a nonlinear equation          *
*                                                     *
* ------------------------------------------------- *
* Arguments                                           *
* Input                                               *
*   A - Left-end point of interval                    *
*   B - Right-end point of interval                   *
*   EPS - Error bound                                 *
* Output                                              *
*   S - Status of completion of task                  *
*   ROOT - Final solution                             *
*   COUNT - Number of iterations done                 *
* ------------------------------------------------- *
* Local Variables                                     *
*   X0,X1,X2,F0,F1,F2                                 *
* ------------------------------------------------- *
* Functions invoked                                   *
*   F,ABS                                             *
* ------------------------------------------------- *
* Subroutines called                                  *
*   NIL                                               *
* ------------------------------------------------- *
```

```
      REAL  A,B,EPS,X0,X1,X2,F0,F1,F2,F,ABS
      INTEGER  S,COUNT
      INTRINSIC  ABS`
      EXTERNAL  F
      X1  =  A
      X2  =  B
      F1  =  F(X1)
      F2  =  F(X2)
* Test  if  A  and  B  bracket  a  root
      IF(F1*F2  .GT.0)  THEN
        S  =  0
        RETURN
      ENDIF
      WRITE(*,*)'      X1          X2'
      COUNT  =  1
111   X0  =  X1  -  F1  *  (X2-X1)/(F2-F1)
      F0  =  F(X0)
      IF(F1*F0  .LT.0)  THEN
        X2  =  X0
        F2  =  F0
      ELSE
        X1  =  X0
        F1  =  F0
      ENDIF
      WRITE(*,*)  X1,X2
* Test  if  desired  accuracy  is  achieved
      IF(ABS((X2-X1)/X2)  .LT.EPS)  THEN
        S  =  1
        ROOT  =  (X1+X2)*0.5
        RETURN
      ELSE
        COUNT  =  COUNT+1
        GO  TO  111
      ENDIF
      END
* -------------- End  of  subroutine  FAL---------------  *
* ------------------------------------------------------  *
* Function  subprogram  F(X)
* ------------------------------------------------------  *
      REAL  FUNCTION  F(X)
      REAL  X
      F  =  X*X+X-2
      RETURN
      END
```

```
* -------------- End of function F(X) ---------------- *
```

### Test Results of FALSE

The program was used to find a root of the equation

$$x^2 + x - 2 = 0$$

using the initial values (1.5, 2.0) and (–3.0, 0.0). Test results are given below:

```
First run

    Input starting values
    1.5 2.0

    SOLUTION BY FALSE POSITION METHOD
    Starting points do not bracket any root
    Stop - Program terminated.

Second run

    Input starting values
    -3.0  0.0

    SOLUTION BY FALSE POSITION METHOD
            X1                       X2
         -3.0000000             -1.0000000
         -3.0000000             -1.6666670
         -3.0000000             -1.9090910
         -3.0000000             -1.9767440
         -3.0000000             -1.9941520
         -3.0000000             -1.9985360
         -3.0000000             -1.9996340
         -3.0000000             -1.9999080
         -3.0000000             -1.9999770
         -3.0000000             -1.9999940
         -3.0000000             -1.9999990
         -3.0000000             -2.0000000
         -3.0000000             -2.0000000
         -2.0000000             -2.0000000
    Root  =  -2.0000000
    F(ROOT)  =  .0000000
    NO.OF ITERATIONS = 14

    Stop - Program terminated.
```

Note that the program outputs a message when the given set of initial values do not bracket any root. When a root is possible, the process of iteration stops when the relative error satisfies the condition

$$\left|\frac{x_n - x_{n-1}}{x_n}\right| < \text{EPS}$$

## 6.8 NEWTON-RAPHSON METHOD

Consider a graph of $f(x)$ as shown in Fig. 6.5. Let us assume that $x_1$ is an approximate root of $f(x) = 0$. Draw a tangent at the curve $f(x)$ at $x = x_1$ as shown in the figure. The point of intersection of this tangent with the $x$-axis gives the second approximation to the root. Let the point of intersection be $x_2$. The slope of the tangent is given by

$$\tan \propto = \frac{f(x_1)}{x_1 - x_2} = f'(x_1) \tag{6.19}$$

where $f(x_1)$ is the slope of $f(x)$ at $x = x_1$. Solving for $x_2$ we obtain

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} \tag{6.20}$$

This is called the *Newton-Raphson formula.*

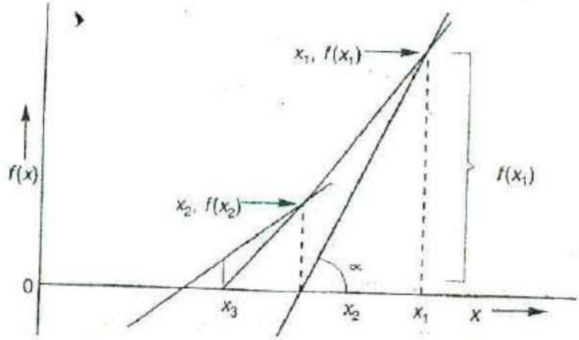


**Fig. 6.5** Newton-Raphson method

The next approximation would be

$$x_3 = x_2 - \frac{f(x_2)}{f'(x_2)}$$

In general,

$$\boxed{x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}} \tag{6.21}$$

This method of successive approximation is called the *Newton-Raphson method.* The process will be terminated when the difference between two successive values is within a prescribed limit.

The Newton-Raphson method approximates the curve of $f(x)$ by tangents. Complications will arise if the derivative $f'(x_n)$ is zero. In such cases, a new initial value for $x$ must be chosen to continue the procedure.

**Example 6.6**

Derive the Newton-Raphson formula using the Taylor series expansion. Assume that $x_n$ is an estimate of a root of the function $f(x)$. Consider a small interval $h$ such that

$$h = x_{n+1} - x_n$$

We can express $f(x_{n+1})$ using Taylor series expansion as follows:

$$f(x_{n+1}) = f(x_n) + f'(x_n)h + f''(x_n)\frac{h^2}{2!} + \dots$$

If we neglect the terms containing the second order and higher derivatives, we get

$$f(x_{n+1}) = f(x_n) + f'(x_n)h$$

If $x_{n+1}$ is a root of $f(x)$, then

$$f(x_{n+1}) = 0 = f(x_n) + f'(x_n)h$$

Then,

$$h = \frac{-f(x_n)}{f'(x_n)} = x_{n+1} - x_n$$

Therefore,

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

## Newton-Raphson Algorithm

Perhaps the most widely used of all methods for finding roots is the Newton-Raphson method. Algorithm 6.4 describes the steps for implementing Newton-Raphson method iteratively.

### Newton-Raphson Method

1. Assign an initial value to $x$, say $x_0$.
2. Evaluate $f(x_0)$ and $f'(x_0)$
3. Find the improved estimate of $x_0$

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

4. Check for accuracy of the latest estimate.

   Compare relative error to a predefined value $E$. If $\left|\frac{x_1 - x_0}{x_1}\right| \le E$ stop; Otherwise continue.
5. Replace $x_0$ by $x_1$ and repeat steps 3 and 4.

**Algorithm 6.4**

**Example 6.7**

Find the root of the equation

$$f(x) = x^2 - 3x + 2$$

in the vicinity of $x = 0$ using Newton-Raphson method.

$$f'(x) = 2x - 3$$

Let $x_1 = 0$ (first approximation)

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}$$

$$= 0 - \frac{2}{-3} = \frac{2}{3} = 0.6667$$

$$x_3 = x_2 - \frac{f(x_2)}{f'(x_2)}$$

Similarly,

$$x_3 = 0.6667 - \frac{0.4444}{-1.6667} = 0.9333$$

$$x_4 = 0.9333 - \frac{0.071}{-1.334} = 0.9959$$

$$x_5 = 0.9959 \quad \frac{0.0041}{-1.0082} = 0.9999$$

$$x_6 = 0.9999 - \frac{0.0001}{-1.0002} = 1.0000$$

Since $f(1.0) = 0$, the root closer to the point $x = 0$ is 1.000.

## Convergence of Newton-Raphson Method

Let $x_n$ be an estimate of a root of the function $f(x)$. If $x_n$ and $x_{n+1}$ are close to each other, then, using Taylor's series expansion, we can state

$$f(x_{n+1}) = f(x_n) + f'(x_n)\,(x_{n+1} - x_n) + \frac{f''(R)}{2}\,(x_{n+1} - x_n)^2 \qquad (6.22)$$

where $R$ lies somewhere in the interval $x_n$ to $x_{n+1}$ and third and higher order have been dropped.

Let us assume that the exact root of $f(x)$ is $x_r$. Then $x_{n+1} = x_r$. Therefore $f(x_{n+1}) = 0$ and substituting these values in equation (6.22), we get

$$0 = f(x_n) + f'(x_n)(x_r - x_n) + \frac{f''(R)}{2}\,(x_r - x_n)^2 \qquad (6.23)$$

We know that the Newton's iterative formula is given by

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Rearranging the terms, we get

$$f(x_n) = f'(x_n)\,(x_n - x_{n+1})$$

Substituting this for $f(x_n)$ in Eq. (6.23) yields

$$0 = f'(x_n)(x_r - x_{n+1}) + \frac{f''(R)}{2}\,(x_r - x_n)^2 \qquad (6.24)$$

We know that the error in the estimate $x_{n+1}$ is given by

$$e_{n+1} = x_r - x_{n+1}$$

Similarly,

$$e_n = x_r - x_n$$

Now, equation (6.24) can be expressed in terms of these errors as

$$0 = f'(x_n)\,e_{n+1} + \frac{f''(R)}{2}e_n^2$$

Rearranging the terms we get,

$$\boxed{e_{n+1} = -\frac{f''(R)}{2f'(x_n)}e_n^2} \qquad (6.25)$$

Equation (6.25) shows that the error is roughly proportional to the square of the error in the previous iteration. Therefore, the Newton-Raphson method is said to have *quadratic convergence*.

## Program NEWTON

The NEWTON shows a FORTRAN program for evaluating a root of non-linear equations by Newton-Raphson method. The program uses two external functions, F and FD and one intrinsic function, ABS. The function F evaluates the actual function at a given value of $x$ and FD evaluates the first derivative of the function at $x$.

The program employs the Algorithm 6.4 and prints out the value of a root (when it is found) and the number of iterations required to obtain the result. It also prints the value of the function at that point to check its accuracy. In case the process does not converge within a specified number of iterations, the program outputs a message accordingly.

```
* ----------------------------------------------------------------- *

      PROGRAM NEWTON
* ----------------------------------------------------------------- *
                                                                    *
* Main program
*    This program finds a root of a nonlinear equation             *
*    by Newton-Raphson method                                       *
* ----------------------------------------------------------------- *
* Functions invoked                                                 *
*    ABS,F,FD                                                       *
* ----------------------------------------------------------------- *
* Subroutines used                                                  *
*    NIL                                                            *
* ----------------------------------------------------------------- *
* Variables used                                                  *
*    X0 - Initial value of x                                      *
*    XN - New value of x                                          *
*    FX - Function value at x                                     *
*    FDX - Value of function derivative at x                      *
*    COUNT - Number of iterations done                            *
* -------------------------------------------------------------- *
* Constants used                                                  *
*    EPS - Error bound                                            *
*    MAXIT - Maximum number of iterations permitted               *
* -------------------------------------------------------------- *
      REAL  X0,XN,FX,FDX,ABS,EPS,F,FD
      INTEGER COUNT, MAXIT
      INTRINSIC ABS
      EXTERNAL F,FD
      PARAMETER(EPS = 0.000001, MAXIT = 100)
      WRITE(*,*) 'Input initial value of x'
      READ(*,*) X0
      WRITE(*,*)
      WRITE(*,*) '   SOLUTION BY NEWTON-RAPHSON METHOD'
      WRITE(*,*)
      COUNT = 1
100   FX = F(X0)
      FDX = FD(X0)
      XN = X0 -FX/FDX
      IF(ABS((XN-X0)/XN) .LT.EPS) THEN
        WRITE(*,*) 'Root = ', XN
        WRITE(*,*) 'Function value = ', F(XN)
        WRITE(*,*) 'Number of iterations = ', CCUNT
        WRITE(*,*)
      ELSE
        X0 = XN
        COUNT = COUNT + 1
        IF(COUNT.LE.MAXIT) THEN
            GO TO 100
        ELSE
            WRITE(*,*)
            WRITE(*,*) 'SOLUTION DOES NOT CONVERGE IN'
            WRITE(*,*) MAXIT, ' ITERATIONS'
            WRITE(*,*)
        ENDIF
      ENDIF
      STOP
      END
* --------------- End of main program --------------- *
* ------------------------------------------------------------- *
* Function subprogram F(x)
* ------------------------------------------------------------- *
      REAL FUNCTION F(X)
      REAL X
      F = X*X+X-2
      RETURN
      END
* -------------- End of function F(X) ---------------- *
* ------------------------------------------------------------- *
* Function subprogram FD(x)
* ------------------------------------------------------------- *
      REAL FUNCTION FD(X)
      REAL X
      FD = 2*X+1
      RETURN
      END
* ------------- End of function FD(X) --------------- *
```

***Test Results of NEWTON*** Given below are the outputs of the test runs of the program NEWTON.

*First run*

```
    Input initial value of x
    0
        SOLUTION BY NEWTON-RAPHSON METHOD
    Root = 1.0000000
    Function value = .0000000
    Number of iterations = 6
    Stop - Program terminated.
```

*Second run*

```
    Input initial value of x
    -1.0
        SOLUTION BY NEWTON-RAPHSON METHOD
    Root = -2.0000000
    Function value = .0000000
    Number of iterations = 6
    Stop - Program terminated.
```

*Third run*

```
    Input initial value of x
    1.0
        SOLUTION BY NEWTON-RAPHSON METHOD
    Root = 1.0000000
    Function value = .0000000
    Number of iterations = 1
    Stop - Program terminated.
```

### Example 6.8

Show, through an example, that the number of correct digits approximately doubles with each iteration in Newton-Raphson method.

Given below is the output of NEWTON program for solving the equation $x^3 - 4x^2 + x + 6 = 0$, using an initial estimate of 5.0.

| *Iteration* | *Estimation* | *Correct digits* |
|---|---|---|
| 1 | 5.000000 | NIL |
| 2 | 4.000000 | NIL |
| 3 | 3.411765 | 1 |
| 4 | 3.114462 | 1 |
| 5 | 3.013215 | 2 |
| 6 | 3.000213 | 4 |
| 7 | 3.000000 | 7 |

This shows that the number of correct digits approximately doubles with each iteration near the root.

## Limitations of Newton-Raphson Method V V P

The Newton-Raphson method has certain limitations and pitfalls. The method will fail in the following situations.

1. Division by zero may occur if $f'(x_i)$ is zero or very close to zero.
2. If the initial guess is too far away from the required root, the process may converge to some other root.
3. A particular value in the iteration sequence may repeat, resulting in an infinite loop. This occurs when the tangent to the curve $f(x)$ at $x = x_{i+1}$ cuts the $x$-axis again at $x = x_i$.

## 6.9 SECANT METHOD

Secant method, like the false position and bisection methods, uses two initial estimates but does not require that they must bracket the root. For example, the secant method can use the points $x_1$ and $x_2$ in Fig. 6.6 as starting values, although they do not bracket the root. Slope of the secant line passing through $x_1$ and $x_2$ is given by

$$\frac{f(x_1)}{x_1 - x_3} = \frac{f(x_2)}{x_2 - x_3}$$

$$f(x_1)\,(x_2 - x_3) = f(x_2)\,(x_1 - x_3)$$

or

$$x_3\,[f(x_2) - f(x_1)] = f(x_2)\,x_1 - f(x_1)x_2$$

Then

$$\boxed{x_3 = \frac{f(x_2)x_1 - f(x_1)\,x_2}{f(x_2) - f(x_1)}} \qquad (6.26)$$

By adding and subtracting $f(x_2)x_2$ to the numerator and rearranging the terms we get

$$\boxed{x_3 = x_2 - \frac{f(x_2)(x_2 - x_1)}{f(x_2) - f(x_1)}} \tag{6.27}$$

Equation (6.27) is known as the *secant formula*. If the secant line represents the linear interpolation polynomial of the function $f(x)$ (with the interpolating points $x_1$ and $x_2$) then $x_3$, which intercepts the $x$-axis, represents the approximate root of $f(x)$.

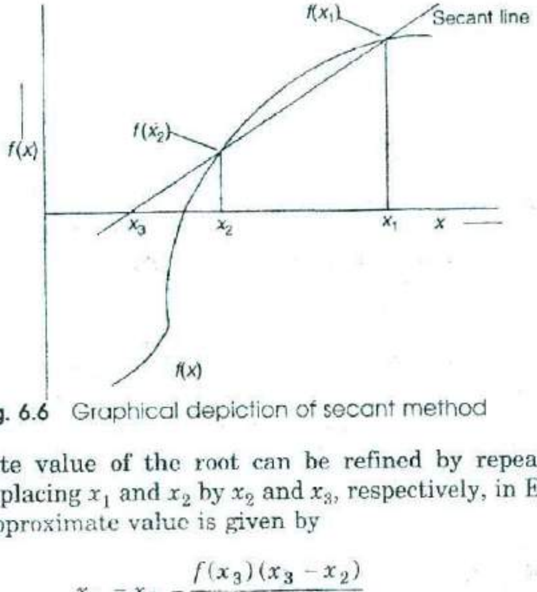


**Fig. 6.6** Graphical depiction of secant method

The approximate value of the root can be refined by repeating this procedure by replacing $x_1$ and $x_2$ by $x_2$ and $x_3$, respectively, in Eq. (6.27). That is, next approximate value is given by

$$x_4 = x_3 - \frac{f(x_3)(x_3 - x_2)}{f(x_3) - f(x_2)}$$

This procedure is continued till the desired level of accuracy is obtained. We can express the secant formula in general form as follows:

$$\boxed{x_{i+1} = x_i - \frac{f(x_i)(x_i - x_{i-1})}{f(x_i) - f(x_{i-1})}} \tag{6.28}$$

Note that Eqs (6.17) and (6.28) are similar and both of them use two initial estimates. However, there is a major difference in their algorithms of implementation. In Eq. (6.17), the latest estimate replaces one of the end points of the interval such that the new interval brackets the root. But, in Eq. (6.28) the values are prelaced in strict sequence, i.e., $x_{i-1}$ is replaced by $x_i$ and $x_i$ by $x_{i+1}$. The points may not bracket the root.

## Secant Algorithm

Note that the value of new approximation of the root depends on the previous two approximations and corresponding functional values. Algorithm 6.5 illustrates how this procedure is implemented to estimate a root with a given level of accuracy.

### Secant Method

```
1. Decide two initial points $x_1$ and $x_2$, accuracy level required, $E$.
2. Compute $f_1 = f(x_1)$ and $f_2 = f(x_2)$
3. Compute $x_3 = \frac{f_2 x_1 - f_1 x_2}{f_2 - f_1}$
4. Test for accuracy of $x_3$.
   If $\left|\frac{x_3 - x_2}{x_3}\right| > E$, then
       set $x_1 = x_2$ and $f_1 = f_2$
       set $x_2 = x_3$ and $f_2 = f(x_3)$
       go to step 3
   otherwise,
       set root = $x_3$
   print results
5. Stop
```

Algorithm 6.5

### Example 6.9

Use the secant method to estimate the root of the equation

$$x^2 - 4x - 10 = 0$$

with the initial estimates of $x_1 = 4$ and $x_2 = 2$

Given $x_1 = 4$ and $x_2 = 2$

$$f(x_1) = f(4) = -10$$

$$f(x_2) = f(2) = -14$$

(Note that these points do not bracket a root)

$$x_3 = x_2 - \frac{f(x_2)(x_2 - x_1)}{f(x_2) - f(x_1)}$$

$$= 2 - \frac{-14(2-4)}{-14-(-10)} = 9.$$

$$= 2 - \frac{-28+56}{-14+10}$$

$$= 2 - \frac{28}{-4}$$

$$= 2 - (-7)$$

$$= 2 + 7 = 9$$

For second iteration,

$$x_1 = x_2 = 2$$

$$x_2 = x_3 = 9$$
$$f(x_1) = f(2) = -14$$
$$f(x_2) = f(9) = 95$$
$$x_3 = 9 - \frac{35(9-2)}{35+14} = 4$$

For third iteration,

$$x_1 = 9$$
$$x_2 = 4$$
$$f(x_1) = f(9) = 95$$
$$f(x_2) = f(4) = -10$$
$$x_3 = 4 - \frac{-10(4-9)}{-10-35} = 5.1111$$

For fourth iteration,

$$x_1 = 4$$
$$x_2 = 5.1111$$
$$f(x_1) = f(4) = -10$$
$$f(x_2) = f(5.1111) = -4.3207$$
$$x_3 = 5.1111 - \frac{-4.3207\,(5.1111-4)}{-4.3207-10} = 5.9563$$

For fifth iteration,

$$x_1 = 5.1111$$
$$x_2 = 5.9563$$
$$f(x_1) = f(5.1111) = -4.3207$$
$$f(x_2) = f(5.9563) = 5.0331$$
$$x_3 = 5.9563 - \frac{5.0331\,(5.9563-5.1111)}{5.0331+4.3207} = 5.5014$$

For sixth iteration,

$$x_1 = 5.9563$$
$$x_2 = 5.5014$$
$$f(x_1) = f(5.9539) = 5.0331$$
$$f(x_2) = f(5.5014) = -1.7392$$
$$x_3 = 5.5014 - \frac{-1.7392(5.5014-5.9563)}{-1.7392+5.0331} = 5.6182$$

The value can be further refined by continuing the process, if necessary.

**Example 6.10**

Compare the secant iterative formula with the Newton formula for estimating a root.

Newton formula: $$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Secant formula: $$x_{n+1} = x_n - \frac{f(x_n)(x_n - x_{n-1})}{f(x_n) - f(x_{n-1})}$$

This shows that the derivative of the function in the Newton formula $f'(x_n)$, has been replaced by the term

$$\frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}$$

in the secant formula. This is a major advantage because there is no need for the evaluation of derivatives. There are many functions whose derivatives may be extremely difficult to evaluate.

However, one drawback of the secant iterative formula is that the previous two iterates are required for estimating the new one. Another drawback of the secant method is its slower rate of convergence. It is proved later in this section that the rate of convergence of secant method is 1.618 while that of the Newton method is 2.

## Convergence of Secant Method

The secant formula of iteration is

$$x_{i+1} = x_i - \frac{f(x_i)(x_i - x_{i-1})}{f(x_i) - f(x_{i-1})} \tag{6.29}$$

Let $x_r$ be actual root of $f(x)$ and $e_i$ the error in the estimate of $x_i$. Then,

$$x_{i+1} = e_{i+1} + x_r$$
$$x_i = e_i + x_r$$
$$x_{i-1} = e_{i-1} + x_r$$

Substituting these in Eq. (6.29) and simplifying, we get the error equation as

$$e_{i+1} = \frac{e_{i-1}f(x_i) - e_i f(x_{i-1})}{f(x_i) - f(x_{i-1})} \tag{6.30}$$

According to the Mean Value Theorem, there exists at least one point, say $x = R_i$, in the interval $x_i$ and $x_r$ such that

$$f'(R_i) = \frac{f(x_i) - f(x_r)}{x_i - x_r}$$

We know that

$$f(x_r) = 0$$

$$x_i - x_r = e_i$$

and therefore

$$f'(R_i) = \frac{f(x_i)}{e_i}$$

or

$$f(x_i) = e_i f'(R_i)$$

Similarly,

$$f(x_{i-1}) = e_{i-1} f'(R_{i-1})$$

Substituting these in the numerator of Eq. (6.30), we get

$$e_{i+1} = e_i e_{i-1} \frac{f'(R_i) - f'(R_{i-1})}{f(x_i) - f(x_{i-1})}$$

That is, we can say

$$\boxed{e_{i+1} \propto e_i e_{i-1}} \tag{6.31}$$

We know that the order of convergence of an iteration process is $p$, if

$$\boxed{e_i \propto e_{i-1}^p} \tag{6.32}$$

or

$$e_{i+1} \propto e_i^p \tag{6.33}$$

Substituting for $e_{i+1}$ and $e_i$ in Eq. (6.31), we get

$$e_i^p \propto e_{i-1}^p \, e_{i-1}$$

or

$$\boxed{e_i \propto e_{i-1}^{(p+1)/p}} \tag{6.34}$$

Comparing the relations (6.32) and (6.31), we observe that

$$p = (p + 1)/p$$

That is,

$$p^2 - p - 1 = 0$$

which has the solutions

$$p = \frac{1 \pm \sqrt{5}}{2}$$

Since $p$ is always positive, we have

$$p = 1.618$$

It follows that the order of convergence of the secant method is 1.618 and the convergence is referred to as *superlinear convergence*.

## Program SECANT

The program SECANT finds a root of a non-linear equation using two initial values supplied. The program employs a function subprogram, F, to evaluate the value of the function and a subroutine subprogram, SEC, to implement the Algorithm 6.5 for estimating the root. The subroutine uses the absolute relative error in the successive approximations for terminating the process.

```
* ---------------------------------------------------------- *
     PROGRAM SECANT
* ---------------------------------------------------------- *
* Main program                                               *
*    This program finds a root of a nonlinear                *
*    equation by secant method                               *
* ---------------------------------------------------------- *
* Functions invoked                                          *
*    F                                                       *
* ---------------------------------------------------------- *
* Subroutines used
*    SEC                                                     *
* ---------------------------------------------------------- *
* Variables used                                             *
*    A - Left endpoint of interval                           *
*    B - Right endpoint of interval                          *
*    ROOT - Final solution                                   *
*    COUNT - Number of iterations completed                  *
* ---------------------------------------------------------- *
* Constants used
*    EPS - Error bound
*    MAXIT - Maximum number of iteration[illegible]
* ---------------------------------------------------------- *
     REAL A,B,ROOT,EPS,F
     INTEGER COUNT,STATUS, MAXIT
     EXTERNAL F,SEC
     PARAMETER(EPS = 0.000001, MAXIT = 50)
     WRITE(*,*)
     WRITE(*,*) '    SOLUTION BY SECANT METHOD'
     WRITE(*,*)
     WRITE(*,*) 'Input two starting points'
     READ(*,*) A,B
     CALL SEC(A,B,X1,X2,EPS,ROOT,COUNT,MAXIT,STATUS)
     IF( STATUS .EQ. 1 ) THEN
       WRITE(*,*)
       WRITE(*,*)'  DIVISION BY ZERO'
       WRITE(*,*)
       WRITE(*,*)' Last X1       =',X1
```

```
      WRITE(*,*)' Last X2      =',X2
      WRITE(*,*)' ITERATIONS =',COUNT
      WRITE(*,*)
    ELSE IF( STATUS .EQ. 2 ) THEN
      WRITE(*,*)
      WRITE(*,*) 'NO CONVERGENCE IN ',MAXIT,' ITERATIONS'
      WRITE(*,*)
    ELSE
      WRITE(*,*)
      WRITE(*,*) 'Root = ', ROOT
      WRITE(*,*) 'Function value at root = ', F(ROOT)
      WRITE(*,*)
      WRITE(*,*) 'Number of iterations = ',COUNT
      WRITE(*,*)
    ENDIF
    STOP
    END
* -------------- End of main program --------------- *
* ------------------------------------------------- *
    SUBROUTINE SEC(A,B,X1,X2,EPS,ROOT,COUNT,MAXIT,STATUS)
* ------------------------------------------------- *
* Subroutine                                         *
*   This subroutine computes a root of an equation   *
*   using the secant method                          *
* ------------------------------------------------- *
* Arguments                                          *
* [redacted]ft end point                             *
              end point                              *
*   EPS       bound                                  *
*   MAXIT - Maximum iterations allowed               *
* Output                                             *
*   X1 - New left point                              *
*   X2 - New right point                             *
*   ROOT - Final solution                            *
*   COUNT - Number of iterations done                *
*   STATUS - Status of completion of the task        *
* ------------------------------------------------- *
* Local Variables                                    *
*   X3,F1,F2,ERROR                                   *
* ------------------------------------------------- *
* Functions invoked                                  *
*   F, ABS                                           *
* ------------------------------------------------- *
* Subroutines called                                 *
*   NIL                                              *
* ------------------------------------------------- *
```

```
      REAL  A,B,X1,X2,X3,EPS,ROOT,F1,F2,F,ABS,ERROR
      INTEGER  COUNT,STATUS,MAXIT
      INTRINSIC  ABS
      EXTERNAL  F
 * Function values at initial points
      X1  =  A
      X2  =  B
      F1  =  F(A)
      F2  =  F(B)
* Compute the root iteratively
      COUNT  =  1
111   IF(ABS(F1-F2)  .LE.  1.E-10)  THEN
        STATUS  =  1
        RETURN
      ENDIF
      X3  =  X2-F2  *  (X2-X1)/(F2-F1)
      ERROR  =  ABS((X3-X2)/X3)
* Test for accuracy
      IF  (ERROR  .GT.  EPS)  THEN
* --- Test  for  convergence
          IF(  COUNT  .EQ.  MAXIT  )  THEN
             STATUS  =  2
             RETURN
        ENDIF
        X1  =  X2
        X2  =  X3
        F1  =  F2
        F2  =  F(X3)
        COUNT  =  COUNT  +  1
        GO  TO  111
*       and  compute  next  approximation
   ENDIF
   ROOT  =  X3
   SATUS  =  3
   RETURN
   END
* -------------- End of subroutine SEC --------------- *
* -------------------------------------------------- *
  Function  subprogram  F(x)
* -------------------------------------------------- *
     REAL  FUNCTION  F(X)
     REAL  X
     F  =  X*X+X-2
```

```
    RETURN
    END
* -------------- End of function F(X) ---------------- *
```

***Test Results of SECANT***

Given below are the outputs of test runs of SECANT

***First run***

```
        SOLUTION BY SECANT METHOD
    Input two starting points
    -3.0001 0
    Root = -2.0000000
    Function value at root = .0000000
    Number of iterations = 11
    Stop - Program terminated.
```

***Second run***

```
        SOLUTION BY SECANT METHOD
    Input two starting points
    0 -3
    Root = -2.0000000
    Function value at root = .0000000
    Number of iterations = 8
    Stop - Program terminated.
```

Note that the program incorporates a test for convergence and also a test for 'division by zero' while evaluating the secant formula (see Eq. 6.27).

## 6.10 FIXED POINT METHOD

Any function in the form of

$$f(x) = 0 \tag{6.35}$$

can be manipulated such that $x$ is on the left-hand side of the equation as shown below

$$x = g(x) \tag{6.36}$$

Equations (6.35) and (6.36) are equivalent and, therefore, a root of Eq. (6.36) is also a root of Eq. (6.35). The root of equation (6.36) is given the point of intersection of the curves $y = x$ and $y = g(x)$. This intersection point is known as the *fixed point* of $g(x)$ (see Fig. 6.7).

The above transformation can be obtained either by algebraic manipulation of the given equation or by simply adding $x$ to both sides of the equation. For example,

$$x^2 + x - 2 = 0$$

can be written as

$$x = 2 - x^2$$

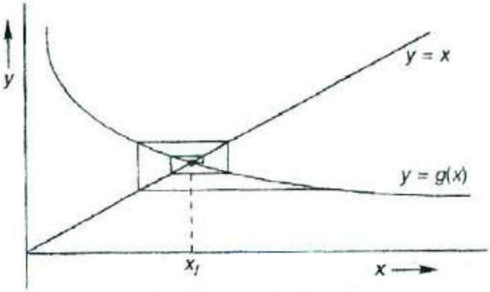


**Fig. 6.7** Fixed point method

or

$$x = x^2 + x - 2 + x = x^2 + 2x - 2$$

Adding of $x$ to both sides is normally done in situations where the original equation is not amenable to algebraic manipulations. For example,

$$\tan x = 0$$

would be put into the form of Eq. (6.36) by adding $x$ to both sides. That is,

$$x = \tan x + x$$

The equation

$$x = g(x)$$

is known as the *fixed point equation*. It provides a convenient form for predicting the value of $x$ as a function of $x$. If $x_0$ is the initial guess to a root, then the next approximation is given by

$$x_1 = g(x_0)$$

Further approximation is given by

$$x_2 = g(x_1)$$

This iteration process can be expressed in general form as

$$\boxed{x_{i+1} = g(x_i) \qquad i = 0,1,2...} \tag{6.37}$$

which is called the *fixed point iteration formula*. This method of solution is also known as the method of *successive approximations* or *method of direct substitution*.

The algorithm is simple. The iteration process would be terminated when two successive approximations agree within some specified error.

### Example 6.11

Locate root of the equation

$$x^2 + x - 2 = 0$$

using the fixed point method.

The given equation can be expressed as

$$x = 2 - x^2$$

Let us start with an initial value of $x_0 = 0$

$$x_1 = 2 - 0 = 2$$

$$x_2 = 2 - 4 = -2$$
$$x_3 = 2 - 4 = -2$$

Since $x_3 - x_2 = 0$, $-2$ is one of the roots of the equation.

Let us assume that $x_0 = -1$. Then

$$x_1 = 2 - 1 = 1$$
$$x_2 = 2 - 1 = 1$$

Another root is 1.

### Example 6.12

Evaluate the square root of 5 using the equation

$$f(x)= x^2 - 5 = 0$$

by applying the fixed point iteration algorithm.

Let us reorganise the function as follows:

$$g(x)= x = 5/x$$

and assume $x_0 = 1$. Then,

$$x_1 = 5$$
$$x_2 = 1$$
$$x_3 = 5$$
$$x_4 = 1$$

The process does not converge to the solution. This type of divergence is known as *oscillatory divergence*.

Let us consider another form of $g(x)$ as shown below:

$$x = x^2 + x - 5$$
$$x_0 = 0$$
$$x_1 = -5$$
$$x_2 = 15$$
$$x_3 = 235$$
$$x_4 = 55455$$

Again it does not converge. Rather it diverges rapidly. This type of divergence is known as *monotone divergence*.

Let us try a third form of $g(x)$.

$$2x = 5/x + x$$

or

$$x = \frac{x + 5/x}{2}$$

$$x_0 = 1$$
$$x_2 = 3$$
$$x_3 = 2.3333$$
$$x_4 = 2.2381$$
$$x_5 = 2.2361$$
$$x_6 = 2.2361$$

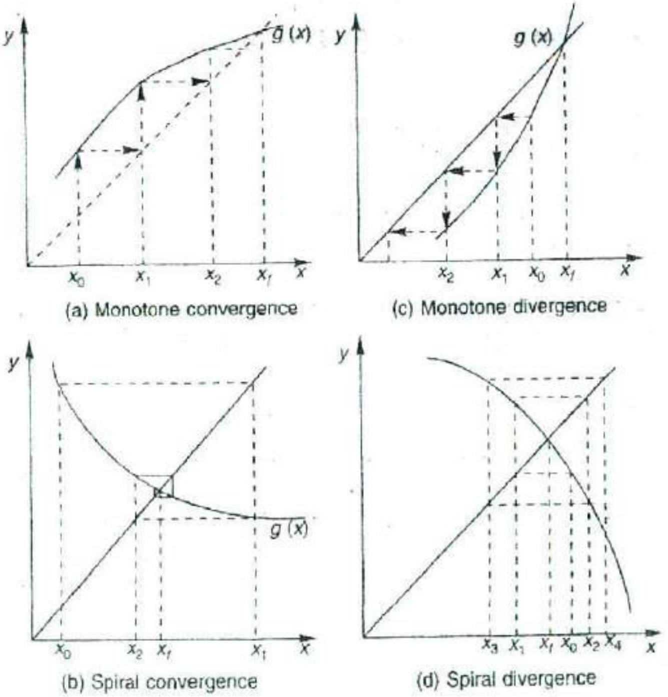
This time, the process converges rapidly to the solution. The square root of 5 is 2.2361.

## Convergence of Fixed Point Iteration

As stated earlier, the iteration function $g(x)$ can be formulated in different forms. Example 6.12 shows that not all forms result in convergence of solution. Convergence of the iteration process depends on the nature of $g(x)$. Figure 6.8 illustrates various patterns of behaviour of the iteration process of the fixed point method. Figures 6.8(a) and 6.8(b), show that the solution converges to the fixed point $x_f$ during the iteration process. However, it does not happen in Fig. 6.8c and 6.8d. Notice that the process converges only when the absolute value of the slope of $y = g(x)$ curve is less than the slope of $y = x$ curve. Since the slope of $y = x$ curve is 1, the necessary condition for convergence is

$$g'(x) < 1$$

We can also notice that, in the neighbourhood of the solution, if the slope of $g(x)$ is positive, the convergence is monotone with "staircase" behaviour, and if the slope of $g(x)$ is negative, the convergence is oscillatory in behaviour. It is also clear that the closer the slope of $g(x)$ is to zero, the faster will be the convergence of the process.



(a) Monotone convergence

(c) Monotone divergence

(b) Spiral convergence

(d) Spiral divergence

**Fig. 6.8** Patterns of behaviour of fixed point iteration process

We can theoretically prove this as follows:
The iteration formula is

$$x_{i+1} = g(x_i) \tag{6.38}$$

Let $x_f$ be a root of the equation. Then,

$$x_f = g(x_f) \tag{6.39}$$

Subtracting equation (6.38) from equation (6.39) yields

$$x_f - x_{i+1} = g(x_f) - g(x_i) \tag{6.40}$$

According to the mean value theorem, there is at least one point, say, $x = R$, in the interval $x_f$ and $x_i$ such that

$$g'(R) = \frac{g(x_f) - g(x_i)}{x_f - x_i}$$

This gives

$$g(x_f) - g(x_i) = g'(R)(x_f - x_i)$$

Substituting this in Eq. (6.40) yields

$$x_f - x_{i+1} = g'(R)(x_f - x_i) \tag{6.41}$$

If $e_i$ represents the error in the $i$th iteration, then Eq. (6.41) becomes

$$e_{i+1} = g'(R)e_i \tag{6.42}$$

This shows that the error will decrease with each iteration only if

$$g'(R) < 1$$

Equation (6.42) implies the following:

1. Error decreases if $g'(R) < 1$
2. Error grows if $g'(R) > 1$
3. If $g'(R)$ is positive, the convergence is monotonic as in Fig. 6.8(a)
4. If $g'(R)$ is negative, the convergence will be oscillatory as in Fig. 6.8(b)
5. The error is roughly proportional to (or less than) the error in the previous step; the fixed point method is, therefore, said to be *linearly convergent*

## Program FIXEDP

The program FIXEDP is the simplest of all programs discussed so far for determining a root of a nonlinear equation. The iteration process is terminated when two successive approximations agree within some specified error. The program uses a control loop to terminate the execution when the process does not converge within a specified number of iterations.

```
* ---------------------------------------------------------- *
      PROGRAM FIXEDP
* ---------------------------------------------------------- *
* Main program                                               *
*    This program finds a root of a function using           *
*    the fixedp point iteration method                       *
* ---------------------------------------------------------- *
* Functions invoked                                          *
*   G, ABS                                                   *
* ---------------------------------------------------------- *
* Subroutines used                                           *
*   NIL                                                      *
* ---------------------------------------------------------- *
* Variables used                                             *
*   X0 - Initial guess                                       *
*   X - Estimated root                                       *
*   ERROR - Relative error                                   *
* ---------------------------------------------------------- *
* Constants used                                             *
*   EPS - Error bound                                        *
*   MAXIT - Maximum Iterations allowed                       *
* ---------------------------------------------------------- *
      REAL X0,X,ERROR,G,ABS,EPS
      INTEGER MAXIT
      INTRINSIC ABS
      EXTERNAL G
      PARAMETER (EPS = 0.00001)

      WRITE (*,*)
      WRITE (*,*) 'SOLUTION BY FIXED POINT ITERATION METHOD'
      WRITE(*,*)

      WRITE(*,*) 'Input initial estimate of root'
      READ (*,*) X0
      WRITE(*,*) 'Maximum iterations allowed'
      READ(*,*) MAXIT

      WRITE(*,*)
      WRITE(*,*)'    ITERATION   VALUE OF X    ERROR'

    DO 100 I = 1, MAXIT
      X = G (X0)
      ERROR = ABS((X-X0)/X)
      WRITE(*,*) I,X,ERROR
      IF (ERROR .LT.EPS) THEN
        WRITE (*,*)
        STOP
      ENDIF
      X0 = X
  100 CONTINUE

      WRITE (*,*) 'Process does not converge to a root'
      write(*,*) 'Exit from loop'

      STOP
      END
* -----------End of main program FIXEDP ----------- *
* ------------------------------------------------------------ *
* Function subprogram G(x)
* ------------------------------------------------------------ *
    REAL FUNCTION G(X)
    REAL X

    G = 2.0-X*X

    RETURN
    END
* -------------- End of function G(X) -------------- *
```

***Test Results of FIXEDP*** The outputs of the program FIXEDP for various initial values are shown below:

*First Run*

```
    SOLUTION BY FIXED POINT ITERATION METHOD
    Input initial estimate of root
    0.0
    Maximum iterations allowed
    10
        ITERATION          VALUE OF X           ERROR
            1               2.0000000         1.0000000
            2              -2.0000000         2.0000000
            3              -2.0000000          .0000000
Stop - Program terminated.
```

*Second Run*

```
    SOLUTION BY FIXED POINT ITERATION METHOD
    Input initial estimate of root
    1
    Maximum iterations allowed
    10
        ITERATION          VALUE OF X           ERROR
            1               1.0000000          .0000000
    Stop - Program terminated.
```

## 6.11 DETERMINING ALL POSSIBLE ROOTS

All the methods discussed so far estimate only one root. What if we are interested in locating all the roots in the given interval? One option is to plot a graph of the function and then identify various independent intervals that bracket the roots. These intervals can be used to locate the various roots.

Another approach is to use an *incremental search* technique covering the entire interval containing the roots. This means that search for a root continues even after the first root is found. The procedure consists of starting at one end of the interval, say, at point $a$, and then searching for a root at every incremental interval till the other end, say, point $b$, is reached (see Fig. 6.9). The end points of each "incremental interval" can

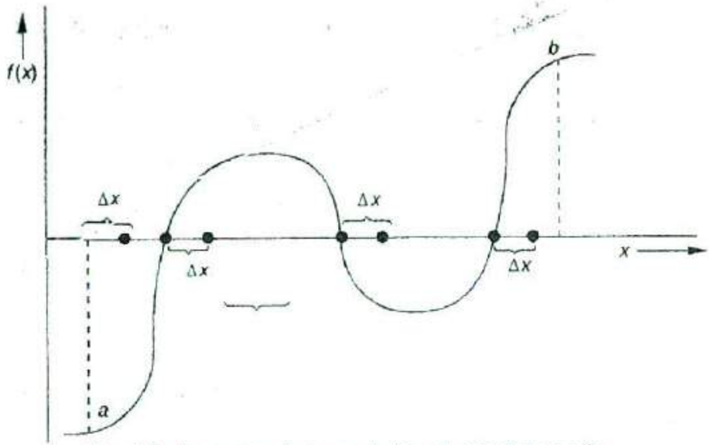


**Fig. 6.9** Incremental search for all possible roots

serve as the initial points for one of the bracketing techniques discussed. Algorithm 6.6 describes the steps for implementing an incremental search technique using the bisection method for locating all roots.

A major problem is to decide the increment size. A small size may mean more iterations and more execution time. If the size is large, then there is a possibility of missing the closely spaced roots.

**Determining all roots**

```
1.  Choose lower limit $a$ and upper limit $b$ of the interval covering all
    the roots.
2.  Decide the size of the incremental interval $\Delta x$
3.  Set $x_1 = a$ and $x_2 = x_1 + x$
4.  Compute $f_1 = f(x_1)$ and $f_2 = f(x_2)$
5.  If $f_1 \times f_2 > 0$, the interval does not bracket any root
       go to step 9
    Otherwise,
       continue
6.  Compute $x_0 = (x_1 + x_2)/2$ and $f_0 = f(x_0)$
7.  If $f_1 \times f_0 < 0$, then
       set $x_2 = x_0$
    else
       set $x_1 = x_0$ and $f_1 = f_0$
8.  If $|(x_2 - x_1)/x_2| < E$, then
       root $= (x_1 + x_2)/2$
       write the value of root
       go to step 9
    else
       go to step 6
9.  If $x_2 < b$, then set $a = x_2$ and go to step 3
10. Stop
```

**Algorithm 6.6**

## 6.12 SYSTEMS OF NONLINEAR EQUATIONS

A system of equations is a set consisting of more than one equation. A system of $n$ equations in $n$ unknown variables is given below.

$$\begin{aligned} f_1(x_1, x_2, \dots x_n) &= 0 \\ f_2(x_1, x_2, \dots x_n) &= 0 \\ &\dots \\ &\dots \\ f_n(x_1, x_2, \dots x_n) &= 0 \end{aligned} \tag{6.43}$$

Equation (6.43) requires values for $x_1, x_2, \dots, x_n$ such that they satisfy all the $n$ equations *simultaneously*. If these equations can be expressed in the form

$$f(x) = a_1x_1 + a_2x_2 + \dots + a_n\, x_n - c = 0$$

then the system is said to be *linear*. On the other hand, if they involve variables with powers, then the system is said to be *nonlinear*.

For example,

$$\begin{aligned} x^2 + 2x - y^2 &= 2 \\ x^2 + 3xy &= 4 \end{aligned}$$

is a system of nonlinear equations in two unknowns. These equations can be expressed in the form of equation (6.43) as

$$f(x, y) = x^2 + 2x - y^2 - 2 = 0 \tag{6.44}$$

$$g(x, y) = x^2 + 3xy - 4 \quad = 0 \tag{6.45}$$

Solution of these equations requires values of $x$ and $y$ that could satisfy both of them simultaneously. We will discuss two methods in this section for solving such equations.

### Fixed Point Method

One simple approach for solving a system of nonlinear equations is to use the fixed point iteration method. Equations (6.11) and (6.15) can be written in the form

$$\begin{aligned} x &= F(x, y) \\ y &= G(x, y) \end{aligned}$$

We can compute $x$ and $y$ using some initial values of $x$ and $y$ on the right-hand side. The new values of $x$ and $y$ can again be used to compute the next set of $x$ and $y$ values. This process can be repeated till a desired level of accuracy in the computed values is reached. This iterative process can be represented in general form as

$$\begin{aligned} x_{i+1} &= F(x_i, y_i) \\ y_{i+1} &= G(x_i, y_i) \end{aligned} \tag{6.46}$$

This can be implemented using the steps given in Algorithm 6.7.

**Fixed point method for a system**

```
1. Define iteration functions
      $F(x, y)$ and $G(x, y)$
2. Decide starting points $x_0$ and $y_0$ and error tolerance $E$
3. $x_1 = F(x_0, y_0)$
   $y_1 = G(x_0, y_0)$
4. If $|x_1 - x_0| < E$ and
      $|y_1 - y_0| < E$, then
         solution obtained;
         go to step 6
5. Otherwise, set
      $x_0 = x_1$
      $y_0 = y_1$
      go to step 3
6. Write values of $x_1$ and $y_1$
7. Stop
```

Algorithm 6.7

### Example 6.13

Solve the following system of nonlinear equations using fixed point method.

$$x^2 - y^2 = 3$$
$$x^2 + xy = 6$$

Iteration functions of these equations are formed as

$$x = y + \frac{3}{x + y}$$

$$y = \frac{6 - x^2}{x}$$

Assume $x_0 = 1$ and $y_0 = 1$

$$x_1 = 2.5$$
$$y_1 = 5$$
$$x_2 = 5.4$$
$$y_2 = -0.1$$
$$x_3 = 0.445$$
$$y_3 = 13$$

The process does not converge. We have to solve the system by forming another set of equations for $x$ and $y$.

Following an approach similar to the one discussed in Section 6.10, it can be shown that the iteration process converges if the following equations are satisfied.

$$\left|\frac{\partial F}{\partial x}\right| + \left|\frac{\partial G}{\partial x}\right| < 1$$

and

$$\left|\frac{\partial F}{\partial y}\right| + \left|\frac{\partial G}{\partial y}\right| < 1 \tag{6.47}$$

The task of forming appropriate iterative functions $F(x, y)$ and $G(x, y)$ to satisfy the above conditions may become very difficult and, therefore, the fixed point iteration process is rarely used to solve systems of nonlinear equations.

### Newton-Raphson Method

The Newton-Raphson method, which was discussed in Section 6.8 for solving single nonlinear equations, can be extended to systems of nonlinear equations. Recall that a first order Taylor series of the form

$$f(x_{i+1}) = f(x_i) + (x_{i+1} - x_i)\, f'(x) \tag{6.48}$$

was used to derive the Newton iteration formula

$$x_{i+1} = x_i - \frac{f(x)}{f'(x)} \tag{6.49}$$

for solving one equation. For the sake of simplicity, let us again consider a two-equation nonlinear system

$$f(x, y) = 0 \qquad g(x, y) = 0$$

First order Taylor series of these equations can be written as

$$f(x_{i+1}, y_{i+1}) = f(x_i, y_i) + (x_{i+1} - x_i)\left|\frac{\partial f_i}{\partial x}\right| + (y_{i+1} - y_i)\left|\frac{\partial f_i}{\partial y}\right| \tag{6.50a}$$

$$g(x_{i+1}, y_{i+1}) = g(x_i, y_i) + (x_{i+1} - x_i)\left|\frac{\partial g_i}{\partial x}\right| + (y_{i+1} - y_i)\left|\frac{\partial g_i}{\partial y}\right| \tag{6.50b}$$

If the root estimates are $x_{i+1}$ and $y_{i+1}$, then

$$f(x_{i+1}, y_{i+1}) = g(x_{i+1}, y_{i+1}) = 0$$

Substituting this in Eq. (6:50) we get the following two linear equations:

$$\Delta x\, f_1 + \Delta y\, f_2 + f = 0 \tag{6.51a}$$

$$\Delta x\, g_1 + \Delta y\, g_2 + g = 0 \qquad \text{(6.51b))}$$

where we denote

$$\Delta x = x_{i+1} - x_i$$

$$\Delta y = y_{i+1} - y_i$$

$$f_1 = \left|\frac{\partial f_i}{\partial x}\right|, \quad f_2 = \left|\frac{\partial f_i}{\partial y}\right|$$

$$g_1 = \left|\frac{\partial g_i}{\partial x}\right|, \quad g_2 = \left|\frac{\partial g_i}{\partial y}\right|$$

$$f = f(x_i, y_i), \quad g = g(x_i, y_i)$$

Solving for $x$ and $y$, we get

$$\Delta x = -\frac{f \cdot g_2 - g \cdot f_2}{f_1 g_2 - f_2 g_1} = -\frac{Dx}{D} \tag{6.52a}$$

$$\Delta y = -\frac{g \cdot f_1 - f \cdot g_1}{f_1 g_2 - f_2 g_1} = -\frac{Dy}{D} \tag{6.52b}$$

where

$$D = \begin{vmatrix} f_1 & f_2 \\ g_1 & g_2 \end{vmatrix} = f_1 g_2 - g_1 f_2$$

is called the *Jacobian matrix*. From Eq. (6.52a) and (6.52b), we can establish the following recurring relations:

$$\boxed{x_{i+1} = x_i - \frac{Dx}{D}} \qquad (6.53(a)$$

$$\boxed{y_{i+1} = y_i - \frac{Dy}{D}} \qquad (6.53b)$$

Equations (6.53a) and (6.53b) are similar to the single-equation Newton formula and may be called the *two-equation* Newton formula. These equations can be used iteratively and simultaneously to solve for the roots of $f(x, y)$ and $g(x, y)$.

Algorithm 6.8 lists the steps involved in implementing the Newton iteration formula for a two-equation system.

**Two equation Newton-Raphson method**

```
1. Define the functions $f$ and $g$
2. Define the Jacobian elements
       $f_1$, $f_2$, $g_1$ and $g_2$
3. Decide starting points $x_0$ and $y_0$ and error tolerance $E$.
4. Evaluate $f$, $g$, $f_1$, $f_2$, $g_1$, $g_2$ at $(x_0, y_0)$
   Compute $Dx$, $Dy$ and $D$
       $x_1 = x_0 - Dx/D$
       $y_1 = y_0 - Dy/D$
```

*(Contd.)*

*(Contd.)*

```
5. Test for accuracy.
   If   $|x_1 - x_0| < E$ and
        $|y_1 - y_0| < E$, then
          solution obtained;
          go to step 7
6. Otherwise, set
          $x_0 = x_1$
          $y_0 = y_1$
          go to step 4
7. Write results
8. Stop
```

**Algorithm 6.8**

**Example 6.18**

Determine the roots of equations

$$x^2 + xy = 6$$
$$x^2 - y^2 = 3$$

using the Newton-Raphson method

Let $F(x, y) = x^2 + xy - 6$

$$G(x, y) = x^2 - y^2 - 3$$

$$f_1 = \frac{\partial F}{\partial x} = 2x + y$$

$$f_2 = \frac{\partial F}{\partial y} = y$$

$$g_1 = \frac{\partial G}{\partial x} = 2x$$

$$g_2 = \frac{\partial G}{\partial y} = -2y$$

Assume the initial guesses as

$$x_0 = 1 \quad \text{and} \quad y_0 = 1$$

*Iteration 1*

$$f_1 = 3, f_2 = 1$$
$$g_1 = 2, g_2 = -2$$

and therefore

$$D = -6 - 2 = -8$$

The values of functions at $x_0$ and $y_0$

$$F = 1^2 + 1 \times 1 - 6 = -4$$
$$G = 1^2 - 1^2 - 3 \quad = -3$$

$$x_1 = 1 - \frac{(-4)(-2) - (-3)(1)}{(-8)} = 2.375$$

$$y_1 = 1 - \frac{(-3)(3)-(-4)(2)}{(-8)} = 0.875$$

*Iteration 2*

$$f_1 = 2 \times 2.375 + 0.875 = 5.625$$
$$f_2 = 0.875$$
$$g_1 = 4.75$$
$$g_2 = -1.75$$
$$F = (2.375)^2 + (2.375)\,(0.875) - 6 = 1.71187$$
$$G = (2.375)^2 - (0.875)^2 = 4.8750$$
$$D = (5.625)\,(-1.75) - (4.75)\,(0.875)$$
$$= -9.8436 - 4.1563 = -14$$
$$x_2 = 2.375 - \frac{(1.7187)(-1.75)-(4.875)(0.875)}{-14}$$
$$= 2.375 - \frac{(-3.0077)-4.2656}{-14} = 2.375 - 0.5195$$
$$= 1.8555$$
$$y_2 = 0.875 - \frac{(4.875)(5.625)-(1.7187)(4.75)}{-14}$$
$$= 0.875 - \frac{27.4218-8.1638}{-14} = 2.2506$$

Continue further to obtain correct answer.

## 6.13 ROOTS OF POLYNOMIALS

We have seen that the methods discussed so far can also be used for evaluation of the roots of polynomials. However, these methods run into problems when the polynomials contain multiple or complex roots. Polynomials are the most frequently used equations in science and engineering and, therefore, require special attention in terms of evaluation of their roots. In this section, we discuss methods to determine all real (not necessarily distinct) and complex roots of polynomials. These methods are specially designed for polynomials and, therefore, cannot be used for transcendental equations.

We will try to use the following properties of $n$th degree polynomials:

1. There are $n$ roots (real or complex)
2. A root may be repeated (multiple roots)
3. Complex roots occur in conjugate pairs
4. If $n$ is odd and all the coefficients are real, then there is at least one real root
5. The polynomial can be expressed as

   $$p(x) = (x - x_r)\, q(x)$$

   where $x_r$ is a root of $p(x)$ and $q(x)$ is the quotient polynomial of order $n - 1$

The number of real roots can be obtained using *Descartes'* rule of sign. This rule states that

1. The number of positive real roots is equal (or less than by an even integer) to the number of sign changes in the coefficients of the equation
2. The number of negative real roots is equal (or less than by an even integer) to the number of sign changes in the coefficients, if $x$ is replaced by $-x$

## Multiple Roots

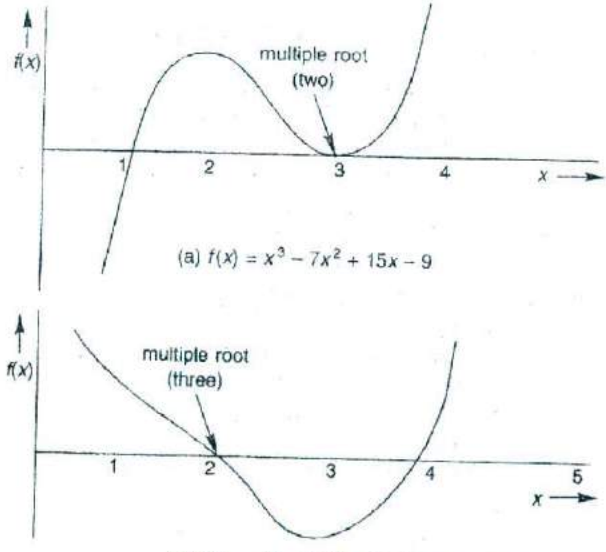
A polynomial function contains a multiple root at a point when the function is tangential to the $x$-axis at that point. For example, the equation

$$x^3 - 7x^2 + 15x - 9 = 0$$

has a double root at $x = 3$ (see Fig. 6.10(a)). The graph is tangent to the $x$-axis at this point. Similarly, the equation

$$x^4 - 10x^3 + 36x^2 - 56x + 32 = 0$$

has a triple root at $x = 2$ (see Fig. 10(b)). Note that the curve crosses the $x$-axis for odd multiple roots and turns back for the even multiple roots. This means that the bracketing methods will have problems in locating the even multiple roots. Another problem is that both $f(x)$ and its derivative $f'(x)$ become zero at the point of multiple roots. As a consequence, the methods (Newton-Raphson and secant) that use derivatives in the denominator might face the problem of division by zero near the roots.



(a) $f(x) = x^3 - 7x^2 + 15x - 9$

(b) $f(x) = x^4 - 10x^3 + 36x^2 - 56x + 32$

**Fig. 6.10** Graph of multiple root polynomials

## Deflation and Synthetic Division

We stated that a polynomial of degree $n$ can be expressed as

$$p(x) = (x - x_r)\, q(x)$$

where $x_r$ is a root of the polynomial $p(x)$ and $q(x)$ is the quotient polynomial of degree $n - 1$. Once a root is found, we can use this fact to obtain a lower degree polynomial $q(x)$ by dividing $p(x)$ by $(x - x_r)$ using a process known as *synthetic division*. The name "synthetic" is used because the quotient polynomial $q(x)$ is obtained without actually performing the division. The activity of reducing the degree of a polynomial is referred to as *deflation*.

The quotient polynomial $q(x)$ can be used to determine the other roots of $p(x)$, because the remaining roots of $p(x)$ are the roots of $q(x)$. When a root of $q(x)$ is found, a further deflation can be performed and the process can be continued until the degree is reduced to one.

Synthetic division is performed as follows:

Let
$$p(x) = \sum_{i=0}^{n} a_i x^i$$

and
$$q(x) = \sum_{i=0}^{n-1} b_i x^i$$

If $p(x) = (x - x_r)\, q(x)$, then

$$\begin{aligned} &a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0 \\ &\quad = (x - x_r)(b_{n-1} x^{n-1} + b_{n-2} x^{n-2} + \dots + b_1 x + b_0) \end{aligned} \tag{6.54}$$

By comparing the coefficients of like powers of $x$ on both the sides of equation (6.54), we get the following relations between them:

$$\begin{aligned} a_n &= b_{n-1} \\ a_{n-1} &= b_{n-2} - x_r\, b_{n-1} \\ &\vdots \\ a_1 &= b_0 - x_r b_1 \\ a_0 &= -x_r\, b_0 \end{aligned}$$

That is

$$a_i = b_{i-1} - x_r\, b_i, \qquad i = n, n-1, \dots 0$$

where $b_n = b_{n-1} = 0$.

Then

$$\boxed{\begin{aligned} b_{i-1} = a_i + x_r\, b_i, \qquad i &= n \dots 1 \\ b_n &= 0 \end{aligned}} \tag{6.55}$$

Equation (6.55) suggests that we can determine the coefficients of $q(x)$ (i.e., $b_{n-1}$, $b_{n-2}$, ... $b_0$) from the coefficients of $p(x)$ (i.e., $a_n$, $a_{n-1}$, ... $a_1$) recursively. Thus, we have obtained the polynomial $q(x)$ without performing any division operation.

### Example 6.15

The polynomial equation

$$p(x) = x^3 - 7x^2 + 15x - 9 = 0$$

has a root at $x = 3$. Find the quotient polynomial $q(x)$ such that

$$p(x) = (x - 3)\, q(x)$$

From $p(x)$, we have

$$a_3 = 1,\ a_2 = -7,\ a_1 = 15, \text{ and } a_0 = -9$$
$$b_3 = 0$$
$$b_2 = a_3 + b_3 \times 3 = 1 + 0 = 1$$
$$b_1 = a_2 + b_2 \times 3 = -7 + 3 = -4$$
$$b_0 = a_1 + b_1 \times 3 = 15 + (-12) = 3$$

Thus the polynomial $q(x)$ is

$$x^2 - 4x + 3 = 0$$

Evaluation of all real roots, including multiple roots, using Newton-Raphson method and synthetic division technique for deflation is presented in Section 6.14.

## Complex Roots

Computing complex roots is much more complex than computing real multiple roots. Recall that complex roots of polynomials with real coefficients occur in conjugate pairs. This suggests that we should isolate the roots of these types by finding the appropriate quadratic factors of the original polynomial (rather than linear factors). Quadratic factors can be obtained by using the process of synthetic division.

Let us assume that

$$h(x) = x^2 - ux - v$$

is an "approximate" quadratic factor of $p(x)$. Then

$$\frac{p(x)}{h(x)} = q(x) + \frac{r(x)}{h(x)} \tag{6.56}$$

where $q(x)$ is the quotient polynomial of degree $(n - 2)$ and $r(x)$ is the remainder. Note that if $h(x)$ is an exact quadratic factor of $p(x)$, then $r(x)$ would be zero. Equation (6.56) can be rewritten as

$$\begin{aligned} p(x) &= q(x)\, h(x) + r(x) \\ &= q(x)\,(x^2 - ux - v) + r(x) \end{aligned} \tag{6.57}$$

Since $q(x)$ is a quotient polynomial, it would be of the form

$$q(x) = b_n\, x^{n-2} + b_{n-1}\, x^{n-3} + \dots + b_2 \tag{6.58}$$

Let us assume that the remainder $r(x)$ takes the form

$$r(x) = b_1\,(x - u) + b_0 \tag{6.59}$$

(The form of $r(x)$ is chosen for the convenience of manipulation).

The objective is to determine the factors $u$ and $v$ such that $r(x)$ becomes zero and, therefore, $h(x)$ becomes an exact factor of $p(x)$ given below.

$$p(x) = a_n\,x^n + a_{n-1}\,x^{n-1} + \ldots + a_1\,x + a_0 \tag{6.60}$$

Substituting Eqs (6.58), (6.59) and (6.60) in Eq. (6.57) and comparing coefficients, we obtain the following relations:

$$\begin{aligned}
b_n &= a_n \\
b_{n-1} &= a_{n-1} + ub_n \\
b_{n-2} &= a_{n-2} + ub_{n-1} + vb_n \\
&\;\;\vdots \\
b_1 &= a_1 + ub_2 + vb_3 \\
b_0 &= a_0 + ub_1 + vb_2
\end{aligned}$$

This can be expressed in general form as

$$\boxed{b_i = a_i + ub_{i+1} + vb_{i+2}} \tag{6.61}$$

where $i = n, n - 1, \ldots 0$

$$b_{n+1} = b_{n+2} = 0$$

Note that all the coefficients $b_i$ are functions of $u$ and $v$ which are unknown.

It is clear that $h(x)$ is a factor of $p(x)$ if and only if

$$\boxed{\begin{aligned} b_1 &= a_1 + ub_2 + vb_3 = 0 \\ b_0 &= a_0 + ub_1 + vb_2 = 0 \end{aligned}} \tag{6.62}$$

Note that Eq. (6.62) is a system of two nonlinear equations in two unknowns, $u$ and $v$. These equations can be solved by using Newton's method discussed in Section 6.14.

Once the values of $u$ and $v$ are known, the roots of the equation

$$x^2 - ux - v$$

can be easily determined using the formula

$$x = \frac{u \pm \sqrt{u^2 + 4v}}{2}$$

The process can be repeated for the quotient polynomial till it becomes either a quadratic or linear polynomial which can be solved for their roots.

## Purification of Roots

*Purification*, as the name indicates, is the process of refining the roots that do not satisfy the required accuracy conditions. These roots may be used again for testing the original problem and improving their approximations.

The Newton-Raphson method is a popular one used for purification of roots. The values of the roots obtained through other methods are used as "initial" input values to the Newton method.

## 6.14 MULTIPLE ROOTS BY NEWTON'S METHOD

As discussed earlier, we can locate all real roots of a polynomial by repeatedly applying Newton-Raphson method and polynomial deflation to obtain polynomials of lower and lower degrees. Algorithm 6.9 gives a step-by-step procedure to achieve this.

Note that the deflation process is performed $(n - 1)$ times where $n$ is the degree of the given polynomial. After $(n - 1)$ deflations, the quotient is a linear polynomial of type

$$a_1 x + a_0 = 0$$

and therefore the final root is given by

$$x_r = -\frac{a_0}{a_1}$$

### Evaluation of Multiple Roots

```
1. Obtain degree and coefficients of polynomial ($n$ and $a_i$)
2. Decide an initial estimate for the first root ($x_0$) and error criterion
Do while $n > 1$
    3. Find the root using Newton-Raphson algorithm:
           $x_r = x_0 - \frac{f(x_0)}{f'(x_0)}$
    4. Root ($n$) = $x_r$
    5. Deflate the polynomial using synthetic division algorithm and make
       the factor polynomial as the new polynomial of order $n - 1$
    6. Set $x_0 = x_r$ (initial value for next root)
End of Do
7. Root (1) = $-a_0/a_1$
8. Stop
```

Algorithm 6.9

### Program MULTIR

The program MULTIR locates all real roots of a polynomial by repeatedly applying the Newton-Raphson method as shown in Algorithm 6.9. To achieve this, the program employs two subroutines: first, the subroutine

NEWTON to find a real root of the polynomial, and second, the subroutine DFLAT to reduce the polynomial degree by one. This process is continued till the degree of the polynomial is reduced to one. This is implemented by the DO loop DO 200 I = N, 2, –1.

The subroutine NEWTON, while evaluating a root, also implements a test for accuracy of the root obtained. In case the required accuracy is not obtained within a specified number of iterations, the execution stops after giving an appropriate message.

```
* ----------------------------------------------------- *
      PROGRAM MULTIR
* ----------------------------------------------------- *
* Main program
*     The program finds all the real roots of           *
*      a polynomial                                     *
* ----------------------------------------------------- *
* Functions invoked                                     *
*     NIL                                               *
* ----------------------------------------------------- *
* Subroutines used                                      *
*     NEWTON                                            *
*     DFLAT                                             *
* ----------------------------------------------------- *
* Variables used
*     N - Degree of polynomial                          *
*     A - Polynomial coefficients A(N+1)                *
*     X0 - Initial guess                                *
*     XR - Root obtained by Newton method               *
*     ROOT - Root Vector                                *
*     STATUS - Solution status                          *
* ----------------------------------------------------- *
* Constants used                                        *
*     EPS - Error bound                                 *
*     MAXIT - Maximum iterations permitted              *
* ----------------------------------------------------- *
      REAL A,X0,XR,ROOT,EPS
      INTEGER N,MAXIT,STATUS
      PARAMETER( EPS=0.000001, MAXIT=50 )
      DIMENSION A(11), ROOT(10)

      WRITE(*,*)
      WRITE(*,*) ' EVALUATION OF MULTIPLE ROOTS
      WRITE(*,*)

      WRITE(*,*) 'Input N, the degree of polynomial"
      READ(*,*) N
      WRITE(*,*) 'Input poly coefficients, A(1) to A(N+1)'
      READ(*,*) (A(I), I=1, N+1)
      WRITE(*,*) 'Input initial guess X'
```

```
    READ(*,*) X0
    WRITE(*,*)
    DO 200 I = N, 2, -1
*     Find I_th root
      CALL NEWTON(N,A,X0,EPS,MAXIT,STATUS,XR)
      IF (STATUS .EQ. 2) THEN
         DO 100 J = N, I+1, -1
100        WRITE(*,*) 'ROOT',J,' =', ROOT(J)
         WRITE(*,*) 'Next root does not converge in'
         WRITE(*,*) MAXIT, ' iterations'
         WRITE(*,*)
         STOP
      ENDIF
      ROOT(I) = XR
*     Deflate the polynomial by division (X - XR)
      CALL DFLAT(N,A,XR)
      X0 = XR
*     Proceed to find next root
200 CONTINUE
* Compute the last root
    ROOT(1) = - A(1)/A(2)
* Write results
    WRITE(*,*) 'ROOTS OF POLYNOMIAL ARE:'
    WRITE(*,*)
    DO 300 I = 1, N
      WRITE(*,*) 'ROOT',I,' =', ROOT(I)
300 CONTINUE
    WRITE(*,*)
    STOP
    END
* ----------- End of main program MULTIR -----------    *
* -----------------------------------------------------  *
  SUBROUTINE NEWTON(N,A,X0,EPS,MAXIT,STATUS,XR)
* -----------------------------------------------------  *
* Subroutine                                             *
*   This subroutine finds a root of the polynomial       *
*   using the Newton-Raphson method                      *
* -----------------------------------------------------  *
* Arguments                                              *
* Input                                                  *
*   N - Degree of polynomial                             *
*   A - Array of polynomial coefficients                 *
*    X0 - Initial guess for a root                              *
*    EPS - Error bound                                          *
*    MAXIT - Maximum iterations permitted                       *
* Output
*    STATUS - Solution status                                   *
*    XR - Root obtained by Newton method                        *
* ------------------------------------------------------------- *
* Local Variables                                               *
*    COUNT - Number of iterations performed                     *
*    FX - Value of polynomial function at X0                    *
*    FDX - Value of function derivative at X0                   *
* ------------------------------------------------------------- *
* Functions invoked                                             *
*    ABS                                                        *
* ------------------------------------------------------------- *
* Subroutines called                                            *
*    NIL                                                        *
* ------------------------------------------------------------- *
     REAL A,X0,EPS,XR,ABS
     INTEGER N,MAXIT,STATUS
     INTRINSIC ABS
     DIMENSION A(11)

     COUNT = 1
* Compute the value of function at X0
100  FX = A(N+1)
     DO 111 I = N, 1, -1
        FX = FX * X0 + A(I)
111  CONTINUE
* Compute the value of derivative at X0
     FDX = A(N+1) * N
     DO 222 I = N, 2, -1
        FDX = FDX * X0 + A(I) * (I-1)
222  CONTINUE
* Compute a root XR
     XR = X0 - FX/FDX
* Test for accuracy
     IF(ABS((XR-X0)/XR).LE.EPS) THEN
        STATUS = 1
        RETURN
     ENDIF
* Test for convergence
     IF(COUNT .LT. MAXIT) THEN
        X0 = XR
        COUNT = COUNT + 1
        GOTO 100
```

```
    ELSE
      STATUS = 2
      RETURN
    ENDIF

    END
* ------------- End of subroutine NEWTON------------- *
* ------------------------------------------------- *
    SUBROUTINE DFLAT(N,A,XR)
* ------------------------------------------------- *
* Subroutine                                        *
*   This subroutine reduces the degree of polynomial *
*   by one using synthetic division                 *
* ------------------------------------------------- *
* Arguments                                         *
* Input                                             *
*   N - Degree of polynomial                        *
*   A - Array of coefficients of input polynomial   *
*   XR - A root of the input polynomial             *
* Output                                            *
*   A - coefficients of the reduced polynomial      *
* ------------------------------------------------- *
* Local Variables                                   *
*   B                                               *
* ------------------------------------------------- *
* Functions invoked                                 *
*   NIL                                             *
* ------------------------------------------------- *
* Subroutines called                                *
*   NIL                                             *
* ------------------------------------------------- *
    REAL A,B,XR
    INTEGER N
    DIMENSION A(11), B(11)
* Evaluate the coefficients of the reduced polynomial

    B(N+1) = 0
    DO 1 1 = N, 1, -1
       B(I) = A(I+1) + XR * B(I+1)
1   CONTINUE
* Change coefficients from B array to A array

    DO   2 I = 1, N+1
       A(I) = B(I)
2   CONTINUE
    RETURN
    END
* -------------- End of subroutine DFLAT     -------- *
```

***Test Results of MULTIR*** The program was tested for evaluating the roots of the equation

$$x^2 - 3x + 2 = 0$$

The results of a test run are given below:

```
     EVALUATION OF MULTIPLE ROOTS

 Input N, the degree of polynomial
 2
 Input poly coefficients, A(1) to A(N+1)
 2 -3 1
 Input initial guess X
 0

 ROOTS OF POLYNOMIAL ARE:

 ROOT 1 = 2.0000000
 ROOT 2 = 1.0000000

 Stop - Program terminated.
```

## 6.15 COMPLEX ROOTS BY BAIRSTOW METHOD

We have discussed in Section 6.13 that complex roots of a polynomial equation can be found by using its quadratic factors. We have also seen that if the polynomial

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \dots a_1 x + a_0$$

is divided by quadratic factor

$$h(x) = x^2 - ux - v$$

then the result is a polynomial

$$q(x) = b_n x^{n-2} + b_{n-1} x^{n-3} + \dots + b_2$$

with a remainder

$$r(x) = b_1(x - u) + b_0$$

The values of coefficients $b_i$ are given by the following recurrence formula:

$$\boxed{\begin{aligned} b_n &= a_n \\ b_{n-1} &= a_{n-1} + ub_n \\ b_i &= a_i + ub_{i+1} + vb_{i+2}, \text{ (for } i = n-2 \text{ to } 0) \end{aligned}} \tag{6.63}$$

We know that in order to make $h(x)$ an exact factor of $p(x)$, $r(x)$ should be zero. This implies that

$$b_1 = b_0 = 0$$

We know from the above recurrence formula that

$$b_1 = a_1 + ub_2 + vb_3 = 0$$

$$b_0 = a_0 + ub_1 + vb_2 = 0$$

The problem now is to find the solution of the system of equations

$$\begin{aligned} b_1(u\,,\,v) &= 0 \\ b_0(u\,,\,v) &= 0 \end{aligned} \tag{6.64}$$

Remember, these are nonlinear equations because coefficients $b_i$ are functions of $u$ and $v$. The strategy used to solve the system of Eqs. (6.64) is known as *Bairstow's method*. The method is similar to the Newton-Raphson approach for solving a two-equation system (discussed in Section 6.12). Using the Taylor series expansion (recall Eq. (6.51)), it can be shown that

$$\begin{aligned} \frac{\partial b_1}{\partial u}\Delta u + \frac{\partial b_1}{\partial v}\Delta v &= -b_1 \\ \frac{\partial b_0}{\partial u}\Delta u + \frac{\partial b_0}{\partial v}\Delta v &= -b_0 \end{aligned} \tag{6.65}$$

To solve these equations, we need partial derivatives of $b_i$ coefficients. Differentiating Eq. (6.63) with respect to $u$, we get

$$\frac{\partial b_i}{\partial u} = b_{i+1} + u\frac{\partial b_{i+1}}{\partial u} + v\frac{\partial b_{i+2}}{\partial u}, i = n - 2 \text{ to } 0 \tag{6.66}$$

$$\frac{\partial b_n}{\partial u} = 0$$

$$\frac{\partial b_{n-1}}{\partial u} = b_n + u\frac{\partial b_n}{\partial u} = b_n$$

For convenience, let us denote

$$c_i = \frac{\partial b_i}{\partial u}$$

Then, we have

$$\boxed{\begin{aligned} c_n &= 0 \\ c_{n-1} &= b_n \\ c_i &= b_{i+1} + u\,c_{i+1} + vc_{i+2}, \quad i = n - 2 \text{ to } 0 \end{aligned}} \tag{6.67}$$

We need the following coefficients of $c_i$

$$\boxed{\begin{aligned} \frac{\partial b_1}{\partial u} &= c_1 \\ \frac{\partial b_0}{\partial u} &= c_0 \end{aligned}}$$

$c_1$ and $c_0$ can be evaluated recursively using Eq. (6.67). Now, differenting Eq. (6.63) with respect to $v$,

$$\frac{\partial b_n}{\partial v} = 0$$

$$\frac{\partial b_{n-1}}{\partial v} = u\,\frac{\partial b_n}{\partial v} = 0$$

$$\frac{\partial b_i}{\partial v} = b_{i+2} + u\frac{\partial b_{i+1}}{\partial v} + v\,\frac{\partial b_{i+2}}{\partial v}, \quad i = n-2 \text{ to } 0 \tag{6.68}$$

If we denote

$$d_i = \frac{\partial b_{i-1}}{\partial v}$$

Then, we have

$$d_n = \frac{\partial b_{n-1}}{\partial v} = 0$$

$$d_{n-1} = \frac{\partial b_{n-2}}{\partial v} = b_n$$

$$d_i = \frac{\partial b_{i-1}}{\partial v} = b_{i+1} + u\,\frac{\partial b_i}{\partial v} + v\frac{\partial b_{i+1}}{\partial v}$$

That is,

$$d_i = b_{i+1} + u\ d_{i+1} + v\ d_{i+2}\,,\ i = n-2 \text{ to } 0 \tag{6.69}$$

We need the following coefficients of $d_i$

$$\boxed{\begin{array}{c} \dfrac{\partial b_1}{\partial v} = d_2 \\[2ex] \dfrac{\partial b_0}{\partial v} = d_1 \end{array}}$$

Again, $d_2$ and $d_1$ can be recursively valuated using equation (6.69).

If we compare Eqs (6.67) and (6.68), it is clear that $d_i$ values are identical to $c_i$ values. That is

$$d_i = c_i \qquad \text{for } i = n \text{ to } 0$$

Then, $d_2 = c_2$ and $d_1 = c_1$. This implies that we need not compute the coefficients $d_i$.

Substituting for partial derivatives in terms of $c$ values in Eq. (6.65) we get

$$c_1\,\Delta u + c_2\,\Delta v = -b_1$$
$$c_0\,\Delta u + c_1\,\Delta v = -b_0$$

Then,

$$\Delta u = -\frac{b_1 c_1 - b_0 c_2}{c_1^2 - c_0 c_2}$$

$$\Delta v = -\frac{b_0 c_1 - b_1 c_0}{c_1^2 - c_0 c_2}$$

Now, given the initial values of $u_0$ and $v_0$, we can estimate the values of $u$ and $v$ using the following recurring relations

$$u_{i+1} = u_i - \frac{b_1 c_1 - b_0 c_2}{c_1^2 - c_0 c_2} \tag{6.70a}$$

$$v_{i+1} = v_i - \frac{b_0 c_1 - b_1 c_0}{c_1^2 - c_0 c_2} \tag{6.70b}$$

Note that the main task in Bairstow's method is the evaluation of $b_i$ and $c_i$ coefficients using the Eqs (6.63) and (6.67). Algorithm 6.10 lists the steps to implement Bairstow's method.

### Complex roots by Bairstow's method

```
1. Get polynomial parameters ($n$ and $a_i$ values)
2. Decide initial estimates, $m_0$ and $v_0$ and stopping criterion
            While $n > 2$ : Do
3. Compute $b_i$ coefficients
4. Compute $c_i$ coefficients
5. Compute
       $D = c_1 \times c_1 - c_0 \times c_2$
       $\Delta u = -(b_1 \times c_1 - b_0 \times c_2)/D$
       $\Delta v = -(b_0 \times c_1 - b_1 \times c_0)/D$
       $u = u_0 + \Delta u$
       $v = v_0 + \Delta v$
6. Test for accuracy of $u$ and $v$. If accuracy is ok, then
      solution obtained;
      go to step 8
7. Otherwise, set
       $u_0 = u$
       $v_0 = v$
   go to step 3
8. Find (complex) roots of $x^2 - ux - v = 0$
   write results
9. Set the coefficients of factor polynomial as $a_i$
       $n = n - 2$
       $a_i = b_{i+2}$ (for $i = n$ to 0)
10. Set next values for $u_0$ and $v_0$
       $u_0 = u$
       $v_0 = v$
            End of While-Do
```

(Contd.)

*(Contd.)*

```
11. If $n = 2$, then
        $u = -a_1/a_2,$
        $v = -a_0/a_2,$
        find (complex) roots
        write results
    else
        single root $= -a_0/a_1$
        write results
12. Stop
```

**Algorithm 6.10**

### Example 6.16

Obtain the quadratic factor of the polynomial

$$p(x) = x^3 + x + 10$$

using Bairstow's method with starting values $u = +1.8$ and $v = -1$

Given

$$a_3 = 1,\ a_2 = 0,\ a_1 = 1,\ a_0 = 10$$

Then

$$b_3 = 1$$

$$b_2 = a_2 + ub_3 = 0 + (+1.8) \times 1 = +1.8$$

$$b_1 = a_1 + ub_2 + vb_3 = 1 + (+1.8)\ (+1.8) + (-4)(1) = 0.24$$

$$\begin{aligned} b_0 &= a_0 + ub_1 + vb_2 \\ &= 10 + (+1.8)\ (0.24) + (-4)(1.8) = 3.232 \end{aligned}$$

$$c_3 = 0$$

$$c_2 = 1$$

$$c_1 = b_2 + uc_2 + vc_3 = +1.8 + (+1.8)(1) + (-4 \times 0) = +3.6$$

$$\begin{aligned} c_0 &= b_1 + uc_1 + v\ c_2 \\ &= 0.24 + (+1.8)\ (+3.6) + (-4 \times 1) = 3.72 \end{aligned}$$

$$D = c_1^2 - c_0\, c_2 = (+3.6)^2 - 3.72 \times 1 = 9.24$$

$$\Delta u = \frac{b_1 c_1 - c_0 c_2}{D}$$

$$= -\frac{(0.24)\,(3.6) - (3.232) \times 1}{9.24} = 0.2563$$

$$\Delta v = -\frac{b_0 c_1 - b_1 c_0}{D}$$

$$= -\frac{(3.232)\,(3.6) - (0.24)\,(3.72)}{9.24} = -1.1616$$

$$u = 1.8 + 0.2563 = 2.0563$$

$$v = -4 - 1.1616 = -5.1616$$

Note that the true values of $u$ and $v$ are 2 and $-5$ respectively. Therefore, the estimated values are close to the true values. These values can be refined by further iterations.

## Program COMPR

The program COMPR can locate all the real and complex roots of an equation. The program COMPR uses Bairstow's method to achieve this. The program logic is detailed in the Algorithm 6.10 and implemented as shown in Fig. 6.11.

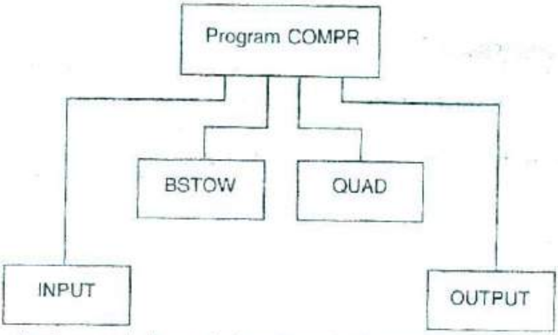


Fig. 6.11 Implementation of algorithm 6.10 to evaluate complex roots

The subprogram INPUT obtains data for polynomial and initial values of the quadratic coefficients. The subprogram BSTOW finds the quadratic factor using multivariable Newton's method and also obtains the reduced polynomial. The subprogram QUAD solves the quadratic equation, the details of which are supplied by BSTOW through the main program COMPR. Finally, the subroutine OUTPUT displays the roots of the quadratic equation.

```
* ---------------------------------------------------------- *
      PROGRAM COMPR
* ---------------------------------------------------------- *
* Main program                                               *
*   This program locates all the roots, both real           *
*   and complex, using Bairstow's method                     *
* ---------------------------------------------------------- *
* Functions invoked                                          *
*   NIL                                                      *
* ---------------------------------------------------------- *
* Subroutines used                                           *
*   INPUT,BSTOW,QUAD,OUTPUT                                  *
* ---------------------------------------------------------- *
* Variables used                                             *
*   N - Degree of polynomial                                 *
*   A - Array of coefficients of polynomial                  *
*   U0,V0 - Initial values of coefficients of the            *
*     quadratic factor                                       *
*   U , V - Computed values of coefficients of the           *
*     quadratic factor                                       *
*   B - Coefficients of the reduced polynomial               *
*   X1,X2 - Roots of the quadratic factor                    *
*   TYPE - Type of roots (real, imaginary or equal)          *
* ---------------------------------------------------------- *
* Constants used                                             *
*   EPS - Error bound                                        *
* ---------------------------------------------------------- *
      INTEGER N, TYPE
      REAL A,B,U0,V0,U,V,X1,X2,EPS,D0,D1,D2
      PARAMETER( EPS = 1.E-6 )
      DIMENSION A(11),B(11)

      WRITE(*,*)
      WRITE(*,*)'EVALUATION OF COMPLEX ROOTS'
      WRITE(*,*)

      CALL INPUT(N,A,U0,V0)
100   IF(N.GT.2) THEN
* ---obtain a quadratic factor
         CALL BSTOW(N,A,B,U0,V0,U,V,EPS)

         D2 = 1
         D1 = -U
         D0 = -V

*---find roots of the quadratic factor
         CALL QUAD(D2,D1,D0,X1,X2,TYPE)

*----print the roots
         CALL OUTPUT(N,TYPE,X1,X2)

*----set the coefficients of the factor polynomial
         N = N-2
         DO 200 I = 1, N+1
            A(I) = B(I+2)
200   CONTINUE

*---set initial values for next quadratic factor
         U0 = U
         V0 = V
         GOTO 100

      ENDIF

      IF(N.EQ.2) THEN
*---polynomial is a quadratic one
         CALL QUAD(A(3),A(2),A(1),X1,X2,TYPE)
         CALL  OUTPUT(N,TYPE,X1,X2)
      ELSE
*----last root of an odd order polynomial
         ROOT = - A(1)/A(2)
         WRITE(*,*)
         WRITE(*,*) 'Final root = ', ROOT
         WRITE(*,*)
   ENDIF
   STOP
   END
* ------------End of main program COMPR ------------- *
* ------------------------------------------------- *
      SUBROUTINE  INPUT(N,A,U0,V0)
* ------------------------------------------------- *
* Subroutine                                         *
*   This subroutine reads polynomial details and     *
*   initial values of the quadratic coefficients    *
* ------------------------------------------------- *
* Arguments                                          *
* Input                                              *
*   NIL                                              *
* Output                                             *
*   N - Degree of polynomial                         *
*   A - Polynomial coefficients                      *
*   U0,V0 - Initial values of the quadratic factor   *
* ------------------------------------------------- *
* Local Variables                                    *
*   NIL                                              *
* ------------------------------------------------- *
* Functions invoked                                  *
*   NIL                                              *
* ------------------------------------------------- *
* Subroutines called                                 *
*   NIL                                              *
* ------------------------------------------------- *
     REAL A,U0,V0
     INTEGER N
     DIMENSION A(11)
     WRITE(*,*)'Input degree of polynomial (N)'
        READ(*,*) N
     WRITE(*,*) 'Input polynomial coefficients A(N+1)
                                              to A(1)'
     DO 11 I = N+1, 1, -1
     READ(*,*) A(I)
11   CONTINUE
     WRITE(*,*) 'Give initial values U0 and V0'
```

```
      READ(*,*) U0,V0
      RETURN
      END
* -------------- End of subroutine INPUT ------------- *
* --------------------------------------------------- *
      SUBROUTINE BSTOW(N,A,B,U0,V0,U,V,EPS)
* --------------------------------------------------- *
* Subroutine                                          *
*   This subroutine finds the quadratic factor using  *
*   multivariable Newton's method and also finds the  *
*   reduced polynomial                                *
* --------------------------------------------------- *
* Arguments                                           *
* Input                                               *
*   N - Degree of polynomial                          *
*   A - Polynomial coefficients                       *
*   U0,V0 - Initial guess for the coefficients
*           of the quadratic factor                   *
*   EPS - Error bound                                 *
* Output                                              *
*   U,V - Computed coefficients of the quadratic      *
*         factor                                      *
*     B - Coefficients of the reduced polynomial      *
* --------------------------------------------------- *
* Local Variables                                     *
*   D,DELU,DELV,C                                     *
* --------------------------------------------------- *
* Functions invoked                                   *
*   ABS                                               *
* --------------------------------------------------- *
* Subroutines called                                  *
*   NIL                                               *
* --------------------------------------------------- *
      INTEGER N
      REAL A,B,U0,V0,U,V,EPS,D,DELU,DELV,C
      INTRINSIC ABS
      DIMENSION A(11), B(11), C(11)

      COUNT = 1
100   B(N+1) = A(N+1)
      B(N) = A(N) + U0 * B(N+1)
      DO 111 I = N-1, 1, -1
        B(I) = A(I) + U0 * B(I+1) + V0 * B(I+2)
111   CONTINUE

      C(N+1) = 0
      C(N) = B(N+1)
      DO 222 I = N-1, 1, -1
        C(I) = B(I+1) + U0 * C(I+1) + V0 * C(I+2)
  222 CONTINUE

      D = C(2) * C(2) - C(1) * C(3)
      DELU = -(B(2) * C(2) - B(1) * C(3))/D
      DELV = -(B(1) * C(2) - B(2) * C(1))/D
      U = U0 + DELU
      V = V0 + DELV
      IF( ABS(DELU/U).LE.EPS .AND. ABS(DELV/V).LE.EPS ) THEN
        RETURN
      ENDIF

      IF(COUNT .LT. 100) THEN
        U0 = U
        V0 = V
        COUNT = COUNT + 1
        GOTO 100
      ELSE
        WRITE(*,*)
        WRITE(*,*)      'NO CONVERGENCE IN 100 ITERATIONS'
        WRITE(*,*)
        STOP
      ENDIF

      END
* ------------- End of subroutine BSTOW ----------- *
* ---------------------------------------------------- *
      SUBROUTINE QUAD(A,B,C,X1,X2,TYPE)
* ---------------------------------------------------- *
* Subroutine                                            *
*   This subroutine solves a quadratic equation of     *
*           2                                           *
*   type AX + BX + C                                    *
* ---------------------------------------------------- *
* Arguments                                             *
* Input                                                 *
*   A,B,C - Coefficients of the quadratic equation     *
* Output                                                *
*   X1,X2 - Roots of the quadratic equation            *
*   TYPE - Type of roots                                *
* ---------------------------------------------------- *
* Local Variables                                       *
*   Q                                                   *
* ---------------------------------------------------- *
* Functions invoked                                     *
*   SQRT,ABS                                            *
* ---------------------------------------------------- *
* Subroutines called                                            *
*   NIL                                                         *
* ------------------------------------------------------------- *
      INTEGER TYPE,IMAGE,EQUAL,UNEQUAL
      REAL A,B,C,X1,X2,SQRT,ABS
      INTRINSIC SQRT,ABS
      PARAMETER( IMAGE = 1, EQUAL = 2, UNEQL = 3)

      Q = B * B - 4 * A * C

      IF(Q.LT.0) THEN
* ------------------Roots are complex
         X1 = -B/(2*A)
         X2 = SQRT(ABS(Q))/(2*A)
         TYPE = IMAGE
      ELSE IF(Q.EQ.0) THEN
* ------------------Roots are real and equal
         X1 = -B/(2*A)
         X2 = X1
         TYPE = EQUAL
      ELSE
* ------------------Roots are real and unequal
         X1 = (-B + SQRT(Q))/(2*A)
         X2 = (-B - SQRT(Q))/(2*A)
         TYPE = UNEQL
      ENDIF

      RETURN
      END

* -------------- End of subroutine QUAD ------------- *
* ------------------------------------------------------------- *
      SUBROUTINE OUTPUT(N,TYPE,X1,X2)
* ------------------------------------------------------------- *
* Subroutine                                                    *
*   This subroutine displays the roots of the                   *
*   quadratic equation                                          *
* ------------------------------------------------------------- *
* Arguments                                                     *
* Input                                                         *
*   N - Degree of the polynomial from which                     *
*       the quadratic factor was obtained                       *
*   TYPE - Type of roots                                        *
*   X1,X2 - Roots of the quadratic factor                       *
* Output                                                        *
*   NIL                                                         *
* ------------------------------------------------------------- *
* Local Variables                                               *
*   NIL                                                         *
* ------------------------------------------------------ *
* Functions invoked                                      *
*   NIL                                                  *
* ------------------------------------------------------ *
* Subroutines called                                     *
*   NIL                                                  *
* ------------------------------------------------------ *
      INTEGER N, TYPE,IMAGE,EQUAL,UNEQL
      REAL X1,X2
      PARAMETER( IMAGE = 1, EQUAL = 2, UNEQL = 3 )
      WRITE(*,*)
      WRITE(*,*) 'Roots of quadratic factor at n = ',N
      WRITE(*,*)
      IF(TYPE .EQ. IMAGE) THEN
        WRITE(*,*) 'Root1 = ', X1, ' + ', X2,'j'
        WRITE(*,*) 'Root2 = ', X1, ' - ', X2,'j'
      ELSE IF(TYPE .EQ. EQUAL) THEN
        WRITE(*,*)    'Root1 = ', X1
        WRITE(*,*)    'Root2 = ', X1
      ELSE
        WRITE(*,*)    'Root1 = ', X1
        WRITE(*,*)    'Root2 = ', X2
      ENDIF
      RETURN
      END
* ------------ End of subroutine OUTPUT ------------ *
```

### Test Results of COMPR

```
    EVALUATION OF COMPLEX ROOTS
Input degree of polynomial (N)
3
Input polynomial coefficients A(N+1) to A(1)
1
0
1
10
Give initial values U0 and V0
1.8 -4.0
Roots of quadratic factor at n =     3
Root1 = 1.0000000 + 2.0000000j
Root2 = 1.0000000 - 2.0000000j
Final root = -2.0000000
Stop - Program terminated.
```

## 6.16 MULLER'S METHOD

Muller's method is an extension of the secant method. Muller's method uses a quadratic curve passing through three points $(x, f(x_1))$, $(x_2, f(x_2))$ and $(x_3, f(x_3))$ as shown in Fig. 6.12 to estimate a root of $f(x)$. One of the roots of the quadratic polynomial $p(x)$ is taken as an approximate value of the root of $f(x)$. As illustrated in Fig. 6.12, the point $x_4$, one of the roots of $p(x)$, is assumed as the next approximation for the root of $f(x)$.

We can write the quadratic polynomial $p(x)$ in the form

$$p(x) = a_0 + a_1 (x - c) + a_2 (x - c)^2 \tag{6.71}$$

Equation (6.71) is known as the shifted-power form of the polynomial and $c$ is a constant known as the centre. If we choose $c = x_3$ then Eq. (6.71) becomes

$$p(x) = a_0 + a_1(x - x_3) + a_2(x - x_3)^2 \tag{6.72}$$

Since $x_4$ is a root of $p(x)$, at $x = x_4$, $p(x) = 0$ and, therefore, Eq. (6.72) becomes

$$a_2 (x_4 - x_3)^2 + a_1(x_4 - x_3) + a_0 = 0$$

Solving the quadratic equation for $(x_1 - x_3)$ we get

$$\boxed{x_4 - x_3 = \frac{-2a_0}{a_1 \pm \sqrt{a_1^2 - 4a_2a_0}}} \tag{6.73}$$

This is one of the forms of quadratic formula, chosen here to minimise error due to any subtractive cancellation. The constants $a_0$, $a_1$ and $a_2$ can be obtained in terms of known function values $f(x_1)$, $f(x_2)$, and $f(x_3)$ as follows:

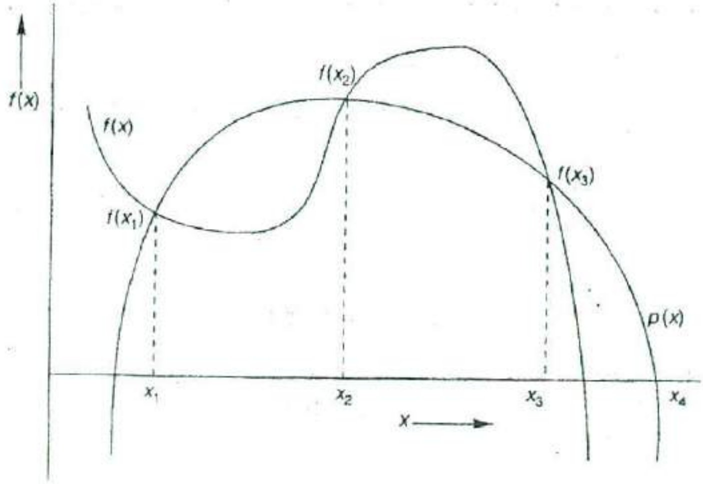


Fig. 6.12 Illustration of Muller's method

At $x = x_1$, $x_2$ and $x_3$, we have

$$a_2(x_1 - x_3)^2 + a_1(x_1 - x_3) + a_0 = p(x_1) = f(x_1)$$
$$a_2(x_2 - x_3)^2 + a_1(x_2 - x_3) + a_0 = p(x_2) = f(x_2)$$
$$a_2(x_3 - x_3)^2 + a_1(x_3 - x_3) + a_0 = p(x_3) = f(x_3)$$

Letting $h_1 = x_1 - x_3$ and $h_2 = x_2 - x_3$, and denoting $f_i = f(x_i)$, we get

$$a_2h_1^2 + a_1h_1 + a_0 = f_1$$
$$a_2h_2^2 + a_1h_2 + a_0 = f_2$$
$$0 + 0 + a_0 = f_3$$

Since $a_0 = f_3$, we can obtain $a_1$ and $a_2$ by solving the equations

$$a_2\,h_1{}^2 + a_1\,h_1 = f_1 - f_3 = d_1$$
$$a_2\,h^2 + a_1\,h_2 = f_2 - f_3 = d_2$$

This results in

$$a_1 = \frac{d_2h_1^2 - d_1h_2^2}{h_1h_2(h_1 - h_2)}$$

$$a_2 = \frac{d_1h_2 - d_2h_1}{h_1h_2(h_1 - h_2)}$$

Equation (6.73) can be written as

$$x_4 = x_3 + h_4$$

where

$$h_4 = \frac{-2a_0}{a_1 \pm \sqrt{a_1^2 - 4a_2a_0}}$$

The sign in the denominator of $h_4$ is chosen such that $h_4$ is as small in magnitude as possible so that $x_3$ is close to $x_4$. That is, the magnitude of $\left(a_1 \pm \sqrt{a_1^2 - 4a_2a_0}\right)$ should be large.

This process is then repeated using $x_2$, $x_3$ and $x_4$ as the initial three points to obtain the next approximation $x_5$.

$$x_5 = x_4 + h_5$$

The process is continued till $f(x_i)$ is within the specified accuracy. Algorithm 6.11 lists the steps in detail for computing a root by Muller's method

**Muller's Method**

```
1. Decide the initial three points and stopping criterion
2. Compute $f_1 = f(x_1)$, $f_2 = f(x_2)$, $f_3 = f(x_3)$
3. Compute
       $h_1 = x_1 - x_3$, $h_2 = x_2 - x_3$
       $d_1 = f_1 - f_3$, $d_2 = f_2 - f_3$
```

*(Contd.)*

*(Contd.)*

```
4. Compute parameters $a_0, a_1, a_2$
       $a_0 = f_3$
       $a_1 = \dfrac{d_2 h_1^2 - d_1 h_2^2}{h_1 h_2 (h_1 - h_2)}$
       $a_2 = \dfrac{d_1 h_2 - d_2 h_1}{h_1 h_2 (h_1 - h_2)}$
5. Compute $h$
       $h = \dfrac{-2a_0}{a_1 \pm \sqrt{a_1^2 - 4a_2 a_0}}$
   (choose the sign in the denominator such that its magnitude is the
   largest. That is, if $a_1$ is positive use + sign, otherwise, – sign)
6. Compute $x_4 = x_3 + h$
7. Compute $f_4 = f(x_4)$
8. If $f(x_4)$ satisfies the given criterion, then
       root is obtained,
       go to step 10
9. Otherwise, set
       $x_1 = x_2,\ x_2 = x_3,\ x_3 = x_4$ and
       $f_1 = f_2,\ f_2 = f_3,\ f_3 = f_4$, then
   go to step 3
10. Write the value of root ($x_4$)
11. Stop
```

**Algorithm 6.11**

### Example 6.17

Solve the Leonardo equation

$$f(x) = x^3 + 2x^2 + 10x - 20 = 0$$

by Muller's method

Let us assume the three starting points as

*Iteration 1*

$x_1 = 0,\ x_2 = 1,\ x_3 = 2$

$$f_1 = -20$$

$$f_2 = -7$$

$$f_3 = 16$$

$$h_1 = x_1 - x_3 = -2$$

$$h_2 = x_2 - x_3 = -1$$

$$d_1 = f_1 - f_3 = -36$$

$$d_2 = f_2 - f_3 = -23$$

$$D = h_1 h_2 (h_1 - h_2)$$

$$= 2(-2 + 1) = -2$$

$$a_1 = \frac{(-23)(-2)^2 - (-36)(-1)^2}{-2} = 28$$

$$a_2 = \frac{(-36)(-1) - (-23)(-2)}{-2} = 5$$

$$h = \frac{-2 \times 16}{28 \pm \sqrt{28^2 - 4(5)(16)}}$$

$$= -\frac{32}{49.540659} \quad \text{(choosing + sign)}$$

$$= -0.645934$$

$$x_4 = x_3 + h = 1.3540659$$

*Iteration 2*

$$x_1 = 1$$

$$x_2 = 2$$

$$x_3 = 1.3540659$$

$$h_1 = x_1 - x_3 = -0.3540659$$

$$h_2 = x_2 - x_3 = 0.645934$$

$$f_1 = -7$$

$$f_2 = 16$$

$$f_3 = f(1.3540659) = -0.3096797$$

$$d_1 = f_1 - f_3 = -6.6903202$$

$$d_2 = f_2 - f_3 = 16.3096797$$

$$D = h_1 h_2 (h_1 - h_2) = 0.2287031$$

$$a_1 = \frac{d_2 h_1^2 - d_1 h_2^2}{D} = 21.145459$$

$$a_2 = \frac{d_1 h_2 - d_2 h_1}{D} = 6.3540717$$

$$a_0 = f_3 = -0.3096797$$

$$h = \frac{-2a_0}{a_1 \pm \sqrt{a_1^2 - 4a_2 a_0}} = \frac{0.6193594}{42.47622} = 0.0145813$$

$$x_4 = x_3 + h = 1.3686472$$

**This process can be continued to obtain better accuracy. The correct answer is 1.368808107.**

### Complex Roots

Note that, in Example 6.17, we obtained a real root of the polynomial. In some cases, we may encounter complex approximations while solving Eq. (6.73). However, in such cases, the imaginary component will normally be small in magnitude and it can be neglected.

In case we are interested in the complex roots as well, we can obtain these by implementing the Muller algorithm using complex arithmetic (which is supported by FORTRAN).

### Multiple Roots

The algorithm can be modified to find more than one root by incorporating the *deflation* procedure using the following equation as discussed in Section 6.13:

$$f'(x) = \frac{f(x)}{x - z_1}$$

### Program MULLER

Design and development of a program to implement Muller's method is left to the reader as an exercise.

## 6.17 SUMMARY

In this chapter, we defined various forms of nonlinear equations and stated a number of approaches to find the roots of such equations. We discussed in detail the following iterative methods to evaluate a root:

- Bisection method (also known as interval halving method)
- False position method (also called linear interpolation method)
- Newton-Raphson method
- Secant method
- Fixed point method (also known as method of direct substitution)
- Muller's method

We also discussed the solution of a system of nonlinear equations using

- Fixed point method
- Newton-Raphson method

We further presented two methods to find the roots of polynomials:

- Newton-Raphson method with synthetic division
- Bairstow's method (for real as well as complex roots)

We discussed the process of converging of iterative methods and proved that

- Newton-Raphson method converges with order of 2
- Bisection method converges linearly
- False position method is linearly convergent
- Secant method follows superlinear convergence

We presented FORTRAN programs and test results for the following methods:

- Bisection method
- False position method
- Newton Raphson method (single root)
- Secant method
- Fixed point method
- Newton-Raphson method (multiple roots)
- Bairstow's method (for complex roots)

## Key Terms

*Algebraic equation*
*Analytical method*
*Bairstow's method*
*Binary chopping method*
*Bisection method*
*Bracketing method*
*Complex number*
*Complex root*
*Convergence*
*Deflation*
*Descartes' rule*
*Direct substitution method*
*Extrapolation method*
*False position method*
*Fixed point equation*
*Fixed point method*
*Graphical method*
*Half-interval method*
*Horner's rule*
*Incremental search*
*Interpolation method*
*Iterative function*
*Iterative method*
*Jacobian matrix*
*Linear*
*Linear interpolation*
*Linearly convergent*
*Monotone convergence*
*Monotone divergence*
*Muller's method*
*Newton-Raphson formula*
*Newton-Raphson method*
*Nonlinear*
*Open end method*
*Polynomial equation*
*Purification*
*Quadratic convergence*
*Quadratic equation*
*Real root*
Regula falsi
*Repeated roots*
*Roots*
*Search bracket*
*Secant formula*
*Secant method*
*Shifted-power form*
*Spiral convergence*
*Spiral divergence*
*Stopping criterion*
*Successive approximations*
*Superlinear convergence*
*Synthetic division*
*Trial and error*
*Transcendental equation*
*Zeros*

## REVIEW QUESTIONS

1. What is a nonlinear equation? Give an example from real-life problems.
2. What is an algebraic equation? Give two examples.

3. Polynomial equations are a simple class of algebraic equations. Explain.
4. What is a transcendental equation? What are its characteristics?
5. What is meant by direct analytical method of solution? What are its limitations?
6. When do we seek the help of graphical method for solving a nonlinear equation?
7. What is an iterative technique? How is it implemented on a computer?
8. Describe the concept applied in the bracketing methods used for solving nonlinear equations.
9. How do we decide initial guess values for solving a polynomial equation using
   (a) open end methods, and
   (b) bracketing methods?
10. What is meant by stopping criterion? State some of the tests that can be used for terminating an iterative process.
11. What is Horner's rule? How does it improve the accuracy of evaluation of a polynomial?
12. Explain the principle of bisection method with the help of an illustration.
13. Explain the principle of false position method.
14. State the Newton-Raphson formula and explain how it is used to obtain a real root.
15. Explain the limitations of using Newton-Raphson method.
16. Note that the secant formula and the false position formula are similar. Then what is the difference between these two methods?
17. How does the secant method compare with the Newton-Raphson method?
18. Discuss the situations where the fixed-point iteration process may not converge to a solution.
19. Describe an algorithm to determine all possible roots of an equation.
20. State the limitations of using the fixed-point approach for solving a system of nonlinear equations.
21. State the Descartes' rule to estimate the number of real roots of a polynomial.
22. What is synthetic division? How is it used to obtain the multiple roots of a polynomial?
23. What is deflation?
24. What is meant by purification of roots? How is it done?
25. Muller's method is an extension of secant method. Explain.
26. Compare, in a tabular form, the order of convergence of various iterative methods used for solving nonlinear equations.

## REVIEW EXERCISES

1. Evaluate the following polynomials using Horner's rule:
   (a) $f(x) = 2x^3 + 4x^2 - 2x + 5$ at $x = 3$
   (b) $f(x) = x^3 - 2x^2 + 5x + 10$ at $x = 5$
   (c) $f(x) = x^4 + 2x^2 - 16x + 5$ at $x = 2$
2. Prove that the bisection method is linearly convergent.
3. How would you decide the two initial values that are required for using the bisection method?
4. Find a root of each of the following equations using the bisection method.
   (a) $e^x - x - 2 = 0$
   (b) $\sin x - 2x + 1 = 0$
   (c) $\log x - \cos x = 0$
   (d) $x \tan x - 1 = 0$
   (e) $x^3 - x - 3 = 0$
   (f) $4x^3 - 2x - 6 = 0$
   (g) $x^4 - 2x^3 - x - 3 = 0$
5. Derive the false position formula for evaluating a root of a nonlinear equation.
6. Use the false position formula repeatedly and obtain roots of the following equations :
   (a) $x - e^{-x} = 0$
   (b) $\sin x - x + 2 = 0$
   (c) $x^3 - 4x^2 + x + 6 = 0$
   (d) $3x^2 + 6x - 45 = 0$
   (e) $4x^3 - 2x - 6 = 0$
7. Derive the Newton-Raphson iterative formula

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

   for solving $f(x) = 0$
8. Show that the Newton-Raphson method converges to solution quadratically.
9. Obtain the Newton's iterative formula for evaluating the square root of a number. Use this formula to find the square root of 3.
10. Derive a recursive formula for finding the $n$th root of a number, say A.
11. Show that Newton's formula for finding the reciprocal of A is

$$x_{n+1} = x_n\,(2 - \mathrm{A}x_n)$$

12. Find the Newton-Raphson formula for the following functions:
    (a) $f(x) = x^2 - 2x - 1$
    (b) $f(x) = x^3 - x - 3$
    (c) $f(x) = x^3 - 3x - 2$
    (d) $f(x) = \cos x$

(e) $f(x) = xe^{-x}$

(f) $f(x) = x \tan x - 1$

13. Apply Newton's method to fi[redacted]he roots of the following equations:

    (a) $e^{-x} - x = 0$

    (b) $\log x - \cos x = 0$

    (c) $\tan x - x = 0$ → M

    (d) $x - 1.5 \sin x - 2.5 = 0$

14. Compute a root of each of the following equations using Newton-Raphson method

    (a) $x^2 - 5x + 6 = 0,$ $x_0 = 5$

    (b) $x^3 - 1.2x^2 + 2x - 2.4 = 0,$ $x_0 = 2$

    (c) $x^3 - 4x^2 + x + 6 = 0,$ $x_0 = 5$

    (d) $x^4 + 3x^3 - 2x^2 - 12x - 8 = 0,$ $x_0 = 1$

    (e) $x^5 - 3x^2 - 100 = 0,$ $x_0 = 2$

15. Derive the secant formula. How is it different from the false position formula.

16. Prove that the rate of convergence of secant method is better than that of bisection method or false position method.

17. Use the secant method to compute a root of the following equations:

    (a) $4x^3 - 2x - 6 = 0$

    (b) $x^2 - 5x + 6 = 0$

    (c) $x \sin x - 1 = 0$

    (d) $e^x - 3x = 0$

    (e) $x - e^x + 2 = 0$

    (f) $x^5 - 3x^2 - 100 = 0$

18. Derive a condition under which the error in the fixed-point iteration method will decrease with each iteration.

19. Use the fixed-point iteration method to evaluate a root of the equation

$$x^2 - x - 1 = 0$$

using the following forms of $g(x)$:

(a) $x = x^2 - 1$

(b) $x = 1 + 2x - x^2$

(c) $x = \frac{1}{2}(1 + 3x - x^2)$

starting with (i) $x_0 = 1$ and (ii) $x_0 = 2$. Discuss the results.

20. Find the square root of 0.75 by writing $f(x) = x^2 - 0.75$ and solving the equation

$$x = x^2 + x - 0.75$$

by the method of fixed-point iteration. Assume an initial value of $x_0 = -0.8$. Try with an initial value of $x_0 = 0.8$. Comment on the results.

21. Use a suitable method to find to three decimal places the roots of the following equations.

    (a) $x^2 - x - 6 = 0$

    (b) $x^2 + 2x - 0.5 = 0$

    (c) $x^2 - 10 \times \log x = 0$

    (d) $x^3 - 2x^2 - 3x + 10 = 0$

22. Solve the system of equations

$$x^2 + y^2 = 5$$
$$x^2 - y^2 = 1$$

using (a) fixed-point method and (b) two equation Newton-Raphson method. Assume $x_0 = 1$ and $y_0 = 1$.

23. Use Newton's method to solve the following systems of equations:

    (a) $3x^2 - 2y^2 = 1$

    $x^2 - 2x + y^2 + 2y = 8$

    (Assume $x_0 = -1$ and $y_0 = 1$)

    (b) $x^3 - y^2 + 1 = 0$

    $x^2 - 2x + y^3 - 2 = 0$

    (Assume $x_0 = 1$ and $y_0 = 1$)

24. The polynomial

$$p(x) = x^3 - 6x^2 + 11x - 6 = 0$$

has a root at $x = 2$. Find the quotient polynomial $q(x)$ such that

$$p(x) = (x - 2)\, q(x)$$

25. A box open at the top is made from a rectangular piece of plywood measuring 5 by 8 metres by removing square pieces from the corners. What will be the size of square pieces removed if the volume of the box is to be 20 cubic metres?

26. The supply and demand functions of a product are

$$Qs = p^2 - 500$$
$$Qd = p^2 - 60p + 1500$$

Determine the market equilibrium price which occurs when $Qs = Qd$.

27. Use Muller's method to find a root of the following equations:

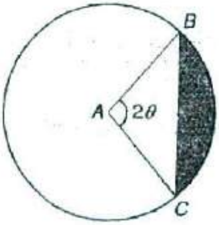
    (a) $x^3 - x - 2$, $x_1 = 1$, $x_2 = 1.2$ and $x_3 = 1.4$

    (b) $1 + 2x - \tan x$, $x_1 = 1.5$, $x_2 = 1.4$ and $x_3 = 1.3$

28. Use Bairstow's method to estimate the roots of

$$f(x) = x^4 - 2x^3 + 4x^2 - 4x + 4$$

29. In the figure shown below, estimate the angle $\theta$ in radians (to two decimal places) using Newton's method (or any other method). Area of triangle $ABC$ equals area shaded.



Also show that there is only one answer in the interval 0 and $\pi/2$.

30. The equation $x \tan x - 1$ occurs in the theory of vibrations.
    (a) How many roots does it have in the interval 0 and $\pi/2$
    (b) Estimate them to two decimal places.

31. The flux equation of an iron core electric circuit is given by

$$f(\phi) = 10 - 2.1\phi - 0.01\phi^3$$

The steady state value of flux is obtained by solving the equation $f(\phi) = 0$. Use a suitable method to estimate the steady state $\phi$.

32. The state of an imperfect gas is given by van der Waals' equation

$$\left(p + \frac{\alpha}{v^2}\right)(v - \beta) = RT$$

or

$$pv^3 - (\beta p + RT)\, v^2 + \alpha v - \alpha\beta = 0$$

Solve the equation for $v$(molar volume) given the following:

$p$ (pressure) = 1.1
$T$ (temperature) = 250° K
$R$ (gas constant ) = 0.082
$\alpha = 3.6$
$\beta = 0.043$

Use any suitable method.

## PROGRAMMING PROJECTS

1. Develop a program to compute all the roots of a polynomial using the bisection method. Use Algorithm 6.6. Test the program for

$$x^3 - 6x^2 + 11x - 6 = 0$$

2. Modify the above program to use Newton-Raphson method instead of bisection method and test the program.
3. Write a program to solve a system of nonlinear equations using
   (a) fixed-point method (Algorithm 6.7)
   (b) Newton-Raphson method (Algorithm 6.8)
4. Write a program for computing a real root of an equation using Muller's method. (Algorithm 6.11).
5. Modify the program in Project 4 to implement the Muller algorithm using complex data type supported in FORTRAN to compute complex roots.
6. Design a menu-driven program to compute a root of a given equation. The menu will provide the choices of methods that a user can select, depending on the nature of equation.

CHAPTER 7

# Direct Solution of Linear Equations

## 7.1 NEED AND SCOPE

Analysis of linear equations is significant for a number of reasons. First, mathematical models of many of the real world problems are either linear or can be approximated reasonably well using linear relationships. Second, the analysis of linear relationship of variables is generally easier than that of nonlinear relationships.

A linear equation involving two variables $x$ and $y$ has the standard form

$$\boxed{ax + by = c} \tag{7.1}$$

where $a$, $b$, and $c$ are real numbers and $a$ and $b$ cannot both equal zero. Notice that the exponent (power) of variables is one. The equation becomes nonlinear if any of the variables has the exponent other than one. Similarly, equations containing terms involving a product of two variables are also considered nonlinear.

Some examples of linear equations are:

$$4x + 7y = 15$$
$$-x - 2/3y = 0$$
$$3u - 2v = -1/2$$

Some examples of nonlinear equations are:

$$2x - xy + y = 2$$
$$x^2 + y^2 = 25$$
$$x + \sqrt{x} = 6$$

In practice, linear equations occur in more than two variables. A ear equation with $n$ variables has the form

$$\boxed{a_1x_1 + a_2x_2 + a_3x_3 + \dots + a_nx_n = b} \tag{7.2}$$

here $a_i$ $(i = 1, 2, \dots n)$ are real numbers and at least one of them is not ero. The main concern here is to solve for $x_i$ $(i = 1, 2, \dots n)$, given the alues of $a_i$ and $b$. Note that an infinite set of $x_i$ values will satisfy the bove equation. There is no unique solution. If we need a unique solution f an equation with $n$ variables (unknowns), then we need a set of $n$ ch independent equations. This set of equations is known as *system of multaneous equations* (or simply, system of equations).

A system of $n$ linear equations is represented generally as

$$\begin{array}{ccccccc}
a_{11}x_1 & + & a_{12}x_2 & + \dots + & a_{1n}x_n & = & b_1 \\
a_{21}x_1 & + & a_{22}x_2 & + \dots + & a_{2n}x_n & = & b_2 \\
\cdot & & \cdot & & \cdot & & \cdot \\
\cdot & & \cdot & & \cdot & & \cdot \\
\cdot & & \cdot & & \cdot & & \cdot \\
a_{n1}x_1 & + & a_{n2}x_2 & + \dots + & a_{nn}x_n & = & b_n
\end{array} \tag{7.3}$$

n matrix notation, Eq. (7.3) can be expressed as

$$Ax = b \tag{7.4}$$

here $A$ is an $n \times n$ matrix, $b$ is an $n$ vector, and $x$ is a vector of $n$ nknowns.

The techniques and methods for solving systems of linear algebraic quations belong to two fundamentally different approaches:

1. Elimination approach
2. Iterative approach

*Elimination approach*, also known as *direct method*, reduces the given system of equations to a form ~~from~~ which the solution can be obtained by simple substitution. We discuss the following elimination methods in this chapter:

1. Basic Gauss elimination method
2. Gauss elimination with pivoting
3. Gauss-Jordan method
4. LU decomposition methods
5. Matrix inverse method

The solution of direct methods do not contain any truncation errors. However, they may contain roundoff errors due to floating point operations.

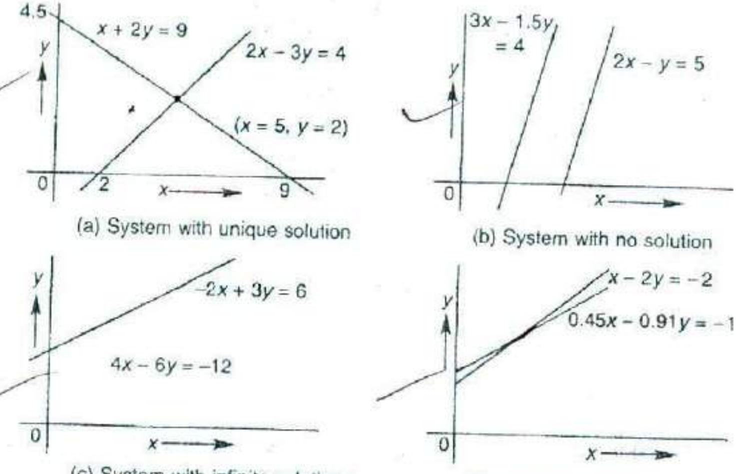
*Iterative approach*, as usual, involves assumption of some initial values which are then refined repeatedly till they reach some accepted level of accuracy. Iterative methods are discussed in Chapter 8.

## EXISTENCE OF SOLUTION

In solving systems of equations, we are interested in identifying values of the variables that satisfy all equations in the system simultaneously.

Given an arbitrary system of equations, it is difficult to say whether the system has a solution or not. Sometimes there may be a solution but it may not be unique. There are four possibilities:

1. System has a unique solution
2. System has no solution
3. System has a solution but not a unique one (i.e., it has infinite solutions)
4. System is ill-conditioned



(a) System with unique solution

(b) System with no solution

(c) System with infinite solutions

(d) Ill-conditioned system

Fig. 7.1 Various forms of a system of two linear equations

## Unique Solution

Consider the system

$$x + 2y = 9$$

$$2x - 3y = 4$$

The system has a solution

$$x = 5 \qquad \text{and} \qquad y = 2$$

Since no other pair of values of $x$ and $y$ would satisfy the equation, the solution is said to be *unique*. The system is illustrated in Fig. 7.1(a).

## No Solution

The equations

$$2x - y = 5$$

$$3x - 3/2y = 4$$

have no solution. These two lines are parallel as shown in Fig. 7.1(b) and, therefore, they never meet. Such equations are called *inconsistent* equations.

### No Unique Solution

The system

$$-2x + 3y = 6$$
$$4x - 6y = -12$$

has many different solutions. We can see that these are two different forms of the same equation and, therefore, they represent the same line (Fig. 7.1(c)). Such equations are called *dependent* equations.

The systems represented in Figures 7.1(b) and 7.1(c) are said to be *singular* systems.

### Ill-Conditioned Systems

There may be a situation where the system has a solution but it is very close to being singular. For example, the system

$$x - 2y = -2$$
$$0.45x - 0.91y = -1$$

has a solution but it is very difficult to identify the exact point at which the lines intersect (Fig. 7.1(d)). Such systems are said to be *ill-conditioned*. Ill-conditioned systems are very sensitive to roundoff errors and, therefore, may pose problems during computation of the solution.

Let us consider a general form of a system of linear equations of size $m \times n$.

$$\begin{array}{c}
a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1 \\
a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = b_2 \\
\cdot \quad \cdot \quad \cdot \quad \cdot \\
\cdot \quad \cdot \quad \cdot \quad \cdot \\
\cdot \quad \cdot \quad \cdot \quad \cdot \\
a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n = b_n
\end{array}$$

In order to effect a unique solution, the number of equations $m$ should by equal to the number of unknowns, $n$. If $m < n$, the system is said to be *under determined* and a unique solution for all unknowns is not possible. On the other hand, if the number of equations is larger than the number of unknowns, then the set is said to be *over determined*, and a solution may or may not exist.

The system is said to be *homogeneous* when the constants $b_i$ are all zero.

## 7.3 SOLUTION BY ELIMINATION

Elimination is a method of solving simultaneous linear equations. This method involves elimination of a term containing one of the unknowns in all but one equation. One such step reduces the order of equations by one. Repeated elimination leads finally to one equation with one unknown. Some rules that are useful in manipulation of the equations are:

1. An equation can be multiplied or divided by a constant.

2. One equation can be added or subtracted from another equation.
3. Equations can be written in any order.

For example, the system

$$2x + y = 4$$
$$5x - 2y = 1$$

can be written in different forms as follows:

1. $4x + 2y = 8$
   $5x - 2y = 1$
2. $-3x + 3y = 3$
   $2x + y = 4$
3. $5x - 2y = 1$
   $2x + y = 4$

Consider a general form of three linear equations:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &= b_2 \\ a_{31}x_1 + a_{82}x_2 + a_{33}x_3 &= b_3 \end{aligned} \tag{7.5}$$

We have three unknowns and three equations. Our objective is to modify this set to the following form:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1 \\ a'_{21}x_1 + a'_{22}x_2 + 0 &= b'_2 \\ a'_{31}x_1 + a'_{32}x_2 + 0 &= b'_3 \end{aligned}$$

This represents a new set of equations with $x_3$ eliminated in the last two equations. The last two equations represent a set with two unknowns.

This system can be further transformed into the form

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1 \\ a''_{21}x_1 + a''_{22}x_2 + 0 &= b''_2 \\ a''_{31}x_1 + 0 + 0 &= b''_3 \end{aligned}$$

Now, the last equation has only one unknown and, therefore, its value can be obtained as

$$x_1 = \frac{b''_3}{a''_{31}}$$

By substituting this in the second equation, we can obtain the value of $x_2$. Finally, $x_3$ can be solved using the computed values of $x_1$ and $x_2$ in the first equation.

Remember that the three-equation system (Eq. (7.5)) can also be transformed into the following form:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1 \\ 0 + a''_{22}x_2 + 0 &= b''_2 \\ a''_{31}x_1 + 0 + a''_{33}x_3 &= b''_3 \end{aligned}$$

Note that a prime indicates that the coefficients have been modified.

The elimination process basically involves the addition of multiples of one equation to other equations so as to set the coefficients of one of the variables in these (other) equations to zero. Example 7.1 illustrates this process.

7-1

Solve the following system of equations by the process of elimination.

$$3x + 2y + z = 10$$
$$2x + 3y + 2z = 14$$
$$x + 2y + 3z = 14$$

The elimination process involves the following steps:

*Step 1: Elimination of x from second and third equations*

Multiply first equation by 2/3 and subtract the result from the second equation. This gives

$$5/3y + 4/3z = 22/3$$

or

$$5y + 4y = 2z$$

Similarly, multiply first equation by 1/3 and subtract the result from the third equation. This gives

$$4y + 8z = 32$$

After step 1, we have the following first derived system:

$$3x + 2y + z = 10$$
$$5y + 4z = 22$$
$$y + 2z = 8$$

*Step 2: Elimination of y from the third equation in the derived system*

Multiply second equation in the derived system by 1/5 and subtract the result from the third. This results in

$$6z = 18$$

The system now has been reduced to an upper triangular form:

$$3x + 2y + z = 10$$
$$5y + 4z = 22$$
$$6z = 18$$

The derivation of this upper triangular system of equations is called the *forward elimination process*.

We can now solve these equations as follows:

$$z = 18/6 = 3$$

Then,

$$5y + 4 \times 3 = 22$$

Therefore,

$$y = (22 - 4 \times 3)/5 = 2$$

Finally,

$$3x + 2 \times 2 + 3 = 10$$
$$x = (10 - 7)/3 = 1$$

Computation of unknowns from the upper triangular system, as illustrated here, is known as *back substitution.*

## 7.4 BASIC GAUSS ELIMINATION METHOD

We have seen in Example 7.1 how to solve a system of three equations using the process of elimination. This approach can be extended to systems with more equations. However, the numerous calculations that are required for larger systems make the method complex and time consuming for manual implementation. Therefore, we need to use computer-based techniques for solving large systems. *Gaussian elimination* is one such technique.

Gauss elimination method proposes a systematic strategy for reducing the system of equations to the upper triangular form using the *forward elimination* approach and then for obtaining values of unknowns using the *back substitution* process. The strategy, therefore, comprises two phases:

1. *Forward elimination phase*: This phase is concerned with the manipulation of equations in order to eliminate some unknowns from the equations and produce an upper triangular system.
2. *Back substitution phase*: This phase is concerned with the actual solution of the equations and uses the back substitution process on the reduced upper triangular system.

Let us consider a general set of $n$ equations in $n$ unknowns:

$$\begin{aligned} a_{11} x_1 + a_{12} x_2 + \dots + a_{1n} x_n &= b_1 \\ a_{21} x_1 + a_{22} x_2 + \dots + a_{2n} x_n &= b_2 \\ \cdot \quad \cdot \quad \cdot \quad &\cdot \\ \cdot \quad \cdot \quad \cdot \quad &\cdot \\ \cdot \quad \cdot \quad \cdot \quad &\cdot \\ a_{n1} x_1 + a_{n2} x_2 + \dots + a_{nn} x_n &= b_n \end{aligned} \tag{7.6}$$

Let us also assume that a solution exists and that it is unique. Algorithm 7.1 illustrates the steps involved in implementing Gauss elimination strategy for such a general system.

**Gauss elimination (basic) method**

1. Arrange equations such that $a_{11} \neq 0$
2. Eliminate $x_1$ from all but the first equation. This is done as follows:
   (i) Normalise the first equation by dividing it by $a_{11}$.
   (ii) Subtract from the second Eq. $a_{21}$ times the normalised first equation.

*(Contd.)*

*(Contd.)*

The result is

$$\left[a_{21}-a_{21}\frac{a_{11}}{a_{11}}\right]x_1+\left[a_{22}-a_{21}\frac{a_{12}}{a_{11}}\right]x_2+\ldots=b_2-a_{21}\frac{b_{11}}{a_{11}}$$

We can see that

$$a_{21}-a_{21}\frac{a_{11}}{a_{11}} \qquad 0$$

Thus, the resultant equation does not contain $x_1$. The new second equation is

$$0+a'_{22}x_2+\ldots+a'_{2n}x_n=b'_2$$

(iii) Similarly, subtract from the third Eq. $a_{31}$ times the normalised first equation.
The result would be

$$0+a'_{32}x_2+\ldots+a'_{3n}x_n=b'_3$$

If we repeat this procedure till the *n*th equation is operated on, we will get the following new system of equations:

$$\begin{aligned} a_{11}x_1+a_{12}x_2+\ldots+a_{1n}x_n&=b_1\\ a'_{22}x_2+\ldots+a'_{2n}x_n&=b'_2\\ \ldots&\\ \ldots&\\ \ldots&\\ a'_{n2}x_2+\ldots+a'_{nn}x_{nn}&=b'_n \end{aligned}$$

The solution of these equations is the same as that of the original equations.

3. Eliminate $x_2$ from the third to the last equation in the new set. Again, we assume that $a'_{22}\neq 0$.
   (i) Subtract from the third equation $a'_{32}$ times the normalised second equation.
   (ii) Subtract from the fourth equation, $a'_{42}$ times the normalised second equation,
   and so on.

   This process will continue till the last equation contains only one unknown, namely, $x_n$. The final form of the equations will look like this:

$$\begin{aligned} a_{11}x_1+a_{12}x_2+\ldots+a_{1n}x_n&=b_1\\ a'_{22}x_2+\ldots+a'_{2n}x_n&=b'_2\\ \ldots&\\ \ldots&\\ a_{nn}^{(n-1)}x_n&=b_n^{(n-1)} \end{aligned}$$

This process is called *triangularisation*. The number of primes indicate the number of times the coefficient has been modified.

*(Contd.)*

*(Contd.)*

4. Obtain solution by back substitution.
   The solution is as follows:

$$x_n = \frac{b_n^{(n-1)}}{a_{nn}^{(n-1)}}$$

This can be substituted back in the $(n-1)^{\text{th}}$ equation to obtain the solution for $x_{n-1}$. This back substitution can be continued till we get the solution for $x_1$.

**Algorithm 7.1**

Note that the relation for obtaining the coefficients of the $k$th derived system has the general form

$$\boxed{a_{ij}^{(k)} = a_{ij}^{(k-1)} - \frac{a_{ik}^{(k-1)}}{a_{kk}^{(k-1)}} a_{kj}^{(k-1)}} \tag{7.7}$$

where

$$i = k+1 \text{ to } n$$
$$j = k+1 \text{ to } n$$
$$a_{ij}^{(0)} = a_{ij} \qquad \text{for } i = 1 \text{ to } n, \quad j = 1 \text{ to } n$$

The $k$th equation, which is multiplied by the factor $a_{ik}/a_{kk}$, is called the *pivot equation* and $a_{kk}$ is called the pivot element. The process of dividing the $k$th equation by $a_{ik}/a_{kk}$ is referred to as *normalisation*.

Similarly, the relation for obtaining the $k$th unknown $x_k$ has the general form

$$\boxed{x_k = \frac{1}{a_{kk}^{(k-1)}} \left[ b_k^{(k-1)} - \sum_{j=k+1}^{n} a_{kj}^{(k-1)} x_j \right]} \tag{7.8}$$

where

$$k = n - 1 \text{ to } 1$$

$$x_n = \frac{b_n^{(n-1)}}{a_{nn}^{(n-1)}}$$

**Example 7.2**

Solve the following 3 × 3 system using the basic Gauss elimination method.

$$3x_1 + 6x_2 + x_3 = 16$$
$$2x_1 + 4x_2 + 3x_3 = 13$$
$$x_1 + 3x_2 + 2x_3 = 9$$

After the first step of elimination using multiplication factor 2/3 and 1/3, we obtain the new system as follows:

$$3x_1 + 6x_2 + x_3 = 16$$
$$0 + 0 + 7x_3 = 7$$
$$0 + 3x_2 + 5x_3 = 11$$

At this point $a_{22} = 0$ and, therefore, the elimination procedure breaks down. We need to reorder the equations as shown below:

$$3x_1 + 6x_2 + x_3 = 16$$
$$3x_2 + 5x_3 = 11$$
$$7x_3 = 7$$

Note that the process of elimination is complete and the solution is:

$$x_3 = 1, x_2 = 2, \text{ and } x_1 = 1$$

## Computational Effort

Computational effort is one of the parameters used to decide the efficiency of a method. Here we estimate the computational effort required in terms of arithmetic operations. The number of operations required for eliminating $x_k$ from the equations below the $k$th row are:

$$\text{Multiplications} : (n - k + 1)(n - k)$$
$$\text{Subtractions} : (n - k + 1)(n - k)$$
$$\text{Divisions} : (n - k + 1)$$

The total operations required in Gauss elimination method is, therefore,

$$\text{Multiplications} = \sum_{k=1}^{n-1} (n - k + 1)(n - k) = \frac{1}{3}n(n^2 - 1)$$

$$\text{Subtractions} = \sum_{k=1}^{n-1} (n - k + 1)(n - k) = \frac{1}{3}n(n^2 - 1)$$

$$\text{Divisions} = \sum_{k=1}^{n} (n - k + 1) = \frac{1}{2}n(n - 1)$$

For back substitution, we are evaluating the $x$ values from $x_n$ to $x_1$. For evaluating the value of $x_k$, we require

$n - k$ multiplications
$n - k$ subtractions
1 division

Therefore, the total operations required for back substitution process are

$$\text{Multiplications} = \sum_{k=1}^{n} (n - k) = \frac{1}{2}n(n - 1)$$

$$\text{Subtractions} = \sum_{k=1}^{n} (n - k) = \frac{1}{2}n(n - 1)$$

$$\text{Divisions} = \sum_{k=1}^{n} 1 = n$$

Total operations required for both the stages are given in Table 7.1.

**Table 7.1** Computational effort required

| | *Elimination process* | *Substitution process* | *Both stages* |
|---|---|---|---|
| Multiplication | $\frac{1}{3}\, n\, (n^2 - 1)$ | $\frac{1}{2}\, n\, (n - 1)$ | $\frac{(n-1)n\,(2n+5)}{6}$ |
| Subtraction | $\frac{1}{3}\, n\, (n^2 - 1)$ | $\frac{1}{2}\, n\, (n - 1)$ | $\frac{(n-1)n\,(2n+5)}{6}$ |
| Division | $\frac{1}{2}\, n\, (n^2 - 1)$ | $n$ | $\frac{n\,(n+1)}{2}$ |

We can thus conclude that the number of multiplications and subtractions grows proportional to $n^3/3$ and the number of divisions proportional to $n^2/2$.

## Program LEG1

The basic Gauss elimination technique enumerated in Algorithm 7.1 is implemented by the program LEG1. The driver program LEG1 uses a separate subprogram GAUSS1 to implement the computational part of the algorithm.

LEG1 obtains the input data from the user and then calls the subprogram GAUSS1 to solve the specified system of linear equations. It finally prints the results when they are received from the subprogram.

The subprogram GAUSS1 receives the details of the equation from the driver program, determines whether the pivot is zero or not, performs the elimination process (if it is not zero), computes $x$ values (by back substitution), and finally sends the results to the driver program.

Note that when the pivot value is near zero, appropriate message is sent to the driver to inform the user accordingly.

```
* ---------------------------------------------------------- *
      PROGRAM LEG1
* ---------------------------------------------------------- *
* Main program                                               *
*   This program solves a system of linear equations         *
*   using simple Gaussian elimination method                 *
* ---------------------------------------------------------- *
* Functions invoked                                          *
*   NIL                                                      *
* ---------------------------------------------------------- *
* Subroutines used                                           *
*   GAUSS1                                                   *
* ---------------------------------------------------------- *
* Variables used                                              *
*   N - Number of equations in the system                     *
*   A - Matrix of coefficients                                *
*   B - Right side vector                                     *
*   X - Solution vector                                       *
* ------------------------------------------------------------*
* Constants used                                              *
*   STATUS - Solution status                                  *
* ------------------------------------------------------------ *
    REAL A,B,X
    INTEGER STATUS,N
    EXTERNAL GAUSS1
    DIMENSION A(10,10), B(10), X(10)
    WRITE(*,*)
    WRITE(*,*) 'SOLUTION BY SIMPLE GAUSS METHOD'
    WRITE(*,*)
    WRITE(*,*) 'What is the size of the system(n)?'
    READ(*,*) N
    WRITE(*,*) 'Input coefficients a(i,j), row-wise,
  +            'one row on each line'
    DO 20 I = 1, N
      READ(*,*) (A(I,J),J=1,N)
20  CONTINUE
    WRITE(*,*) 'Input vector b'
    READ(*,*) (B(I), I = 1, N)
    CALL GAUSS1(N,A,B,X,STATUS)
    IF(STATUS .NE. 0) THEN
      WRITE(*,*)
      WRITE(*,*)   'SOLUTION VECTOR X'
      WRITE(*,*)
      WRITE(*,*)   (X(I), I = 1, N)
      WRITE(*,*)
    ELSE
      WRITE(*,*)
      WRITE(*,*)   'SINGULAR MATRIX, NO SOLUTION'
      WRITE(*,*)   'REORDER EQUATIONS'
      WRITE(*,*)
    ENDIF
    STOP
    END
* ------------- End of main program LEG1 ------------          *
* ----------------------------------------------------          *
    SUBROUTINE GAUSS1(N,A,B,X,STATUS)
* ----------------------------------------------------          *
* Subroutine                                                  *
*   This subroutine solves a set of n linear                  *
*   equations by Gauss elimination method                     *
* ----------------------------------------------------------- *
* Arguments                                                   *
* Input                                                       *
*      N - Number of equations                                *
*      A - Matrix of coefficients                             *
*      B - Right side vector                                  *
* Output                                                      *
       X - Solution vector                                    *
*      STATUS - Solution status                               *
* ----------------------------------------------------------- *
* Local Variables                                             *
*   PIVOT, FACTOR, SUM                                        *
* ----------------------------------------------------------- *
* Functions invoked                                           *
*   NIL                                                       *
* ----------------------------------------------------------- *
* Subroutines called                                          *
*   NIL                                                       *
* ----------------------------------------------------------- *

      REAL A,B,X,PIVOT,FACTOR,SUM
      INTEGER STATUS,N
      DIMENSION A(10,10), B(10), X(10)
* ---------------- Elimination begins ---------------- *

      DO 33 K = 1, N-1
        PIVOT = A(K,K)
        IF(PIVOT .LT.   0.000001) THEN
           STATUS = 0
           RETURN
      ENDIF
      STATUS = 1
      DO 22 I = K+1, N
        FACTOR = A(I,K)/PIVOT
        DO 11 J = K+1, N
              A(I,J) = A(I,J) - FACTOR * A(K,J)
 11     CONTINUE
              B(I) = B(I) - FACTOR * B(K)
 22     CONTINUE
 33   CONTINUE
* ------------- Back substitution begins ------------ *

      X(N) = B(N)/A(N,N)
      DO 55 K = N-1,1,-1
         SUM = 0
         DO 44 J = K+1,N
```

```
            SUM = SUM + A(K,J) * X(J)
44        CONTINUE
          X(K) = (B(K) - SUM)/A(K,K)
55      CONTINUE
        RETURN
        END
* ------------- End of subroutine GAUSS1 ------------- *
```

***Test Run Results***

```
           SOLUTION BY SIMPLE GAUSS METHOD
What is the size of the system(n)?
3
Input coefficients a(i,j), row-wise, one row on each line
2 1 3
4 4 7
2 5 9
Input vector b
1 1 3
SOLUTION VECTOR X
-5.000000E-001      -1.0000000      1.0000000
Stop - Program terminated.
```

## 7.5 GAUSS ELIMINATION WITH PIVOTING

In the basic Gauss elimination method, the element $a_{ij}$ when $i = j$ is known as a pivot element. Each row is normalised by dividing the coefficients of that row by its pivot element. That is

$$a_{kj} = \frac{a_{kj}}{a_{kk}} \qquad j = 1,...,n$$

If $a_{kk} = 0$, $k$th row cannot be normalised. Therefore, the procedure fails. One way to overcome this problem is to interchange this row with another row below it which does not have a zero element in that position (see Example 7.2).

From the given set of equations, it is possible to reorder the equations such that $a_{11}$ is not zero. But subsequently, the values of $a_{kk}$ are continuously modified during the elimination process and, therefore, it is not possible to predict their values beforehand.

The reordering of the rows is done such that $a_{kk}$ of the row to be normalised is not zero. There may be more than one non-zero values in the $k$th column below the element $a_{kk}$. The question is: which one of them is to be selected? It can be proved that roundoff errors would be reduced if the absolute value of the pivot element is large. Therefore, it is suggested that the row with zero pivot element should be interchanged with the row having the largest (absolute value) coefficient in that position. In general, *the reordering of equations is done to improve accuracy, even if the pivot element is not zero.*

The procedure of reordering involves the following steps:

1. Search and locate the largest absolute value among the coefficients in the first column
2. Exchange the first row with the row containing that element
3. Then eliminate the first variable in the second equation as explained earlier
4. When the second row becomes the pivot row, search for the coefficients in the second column from the second row to the $n$th row and locate the largest coefficient. Exchange the second row with the row containing the large coefficient
5. Continue this procedure till $(n - 1)$ unknowns are eliminated.

This process is referred to as *partial pivoting*. There is an alternative scheme known as *complete pivoting* in which, at each stage, the largest element in any of the remaining rows is used as the pivot. Figure 7.2 illustrates the partial and complete pivoting strategies. Algorithm 7.2 shows the implementation steps for partial pivoting.

Complete pivoting requires a lot of overhead and, therefore, it is not generally used (though it may yield slightly improved numerical stability).

### Gauss eliminating with partial pivoting

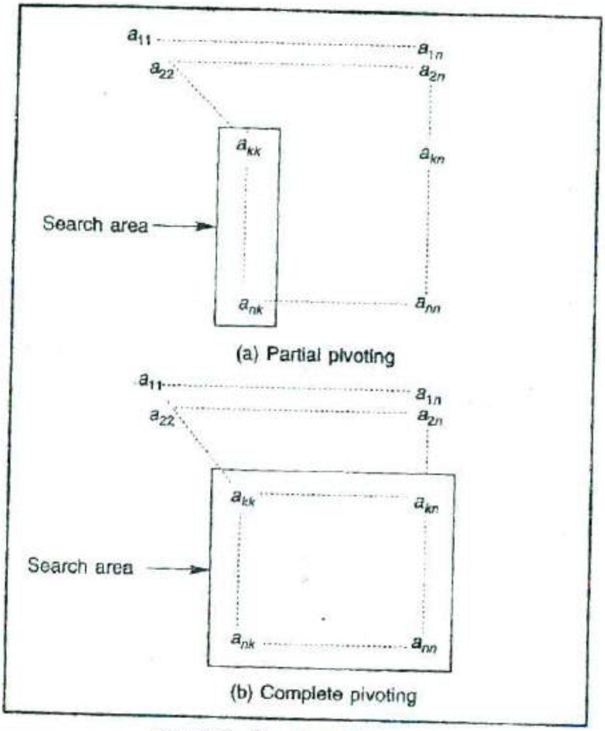
```
1. Input $n$, $a_{ij}$ and $b_i$ values.
2. Beginning from the first equation,
     (i) check for the pivot element
    (ii) if it is the largest among the elements below it, obtain the
         derived system
   (iii) otherwise, identify the largest element and make it the pivot
         element
    (iv) interchange the original pivot equation with the one contain-
         ing the largest element so that
         the later becomes the new pivot equation
     (v) obtain the derived system
    (vi) continue the process till the system is reduced to triangular
         form
3. Compute $x_i$ values by back substitution.
4. Print results.
```

Algorithm 7.2

### Example 7.2

Solve the following system of equations using partial pivoting technique

$$\begin{aligned} 2x_1 + 2x_2 + x_3 &= 6 \\ 4x_1 + 2x_2 + 3x_3 &= 4 \\ x_1 + x_2 + x_3 &= 0 \end{aligned}$$



(a) Partial pivoting

(b) Complete pivoting

**Fig. 7.2** Pivoting strategies

The forward elimination process using partial pivoting is shown below in tabular form. The process involves two steps of elimination and, in both the steps, the rows are interchanged. Note that the absolute value of –3/2 is greater than 1.

| | | | | | |
|---|---|---|---|---|---|
| Original system | 2 | 2 | 1 | 6 | Interchange |
| | 4 | 2 | 3 | 4 | |
| | 1 | –1 | 1 | 0 | |
| Modified original system | 4 | 2 | 3 | 4 | pivot |
| | 2 | 2 | 1 | 6 | |
| | 1 | –1 | 1 | 0 | |
| First derived system | 4 | 2 | 3 | 4 | |
| | | 1 | –1/2 | 4 | Interchange |
| | | –3/2 | 1/4 | –1 | |
| Modified first derived system | 4 | 2 | 3 | 4 | |
| | | –3/2 | 1/4 | –1 | pivot |
| | | 1 | –1/2 | 4 | |

| | | | | |
|---|---|---|---|---|
| Second and final derived system | 4 | 2 | 3 | 4 |
| | | −3/2 | 1/4 | −1 |
| | | | −1/3 | 10/3 |

The solution is

$$x_3 = -10$$
$$x_2 = -1$$
$$x_1 = 9$$

## Program LEG2

Program LEG2 is designed to solve a system of linear equations using Gauss elimination with partial pivoting. The modular structure of the program is shown in Fig. 7.3.

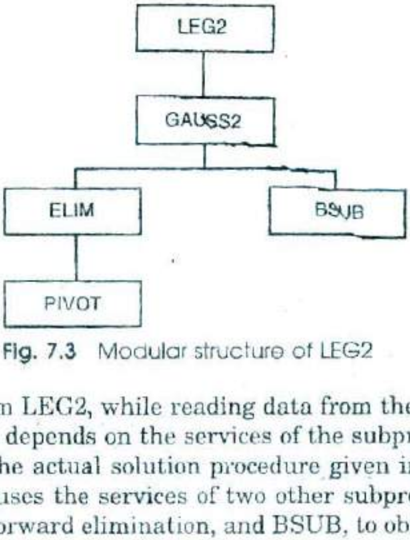


**Fig. 7.3** Modular structure of LEG2

The master program LEG2, while reading data from the user and printing solution vector, depends on the services of the subprogram GAUSS2 for implementing the actual solution procedure given in Algorithm 7.2. GAUSS2, in turn, uses the services of two other subprograms, namely, ELIM, to perform forward elimination, and BSUB, to obtain the solution vector using the back substitution approach.

The subprogram PIVOT undertakes the task of partial pivoting by identifying the pivot element and then rearranging the rows such that the equation containing the pivot element becomes the pivot equation.

```
* ------------------------------------------------------------ *
      PROGRAM LEG2
* ------------------------------------------------------------ *
* Main program                                                 *
*   This program solves a system of linear equations           *
*   using Gaussian elimination with partial pivoting           *
* ------------------------------------------------------------ *
* Functions invoked                                            *
*   NIL                                                        *
* ------------------------------------------------------------ *
* Subroutines used                                           *
*   Gauss2                                                   *
* ---------------------------------------------------------- *
* Variables used                                             *
*   N - Number of equations                                  *
*   A - Coefficients matrix                                  *
*   B - Right side vector                                    *
*   X - Solution vector                                      *
* ---------------------------------------------------------- *
* Constants used                                             *
*   NIL                                                      *
* ---------------------------------------------------------- *
      REAL A,B,X
      INTEGER N
      EXTERNAL GAUSS2
      DIMENSION A(10,10), B(10), X(10)
      WRITE(*,*)
      WRITE(*,*) ' GAUSS METHOD WITH PARTIAL PIVOTING'
      WRITE(*,*)
      WRITE(*,*) 'What is the size of the system(n)?'
      READ(*,*) N
      WRITE(*,*) 'Input coefficients a(i,j), row-wise'
      WRITE(*,*) 'one row on each line'
      DO 20 I = 1, N
        READ(*,*) (A(I,J),J=1,N)
20    CONTINUE
      WRITE(*,*) 'Input vector b'
      READ(*,*) (B(I), I = 1, N)
      CALL GAUSS2(N,A,B,X)
      WRITE(*,*)
      WRITE(*,*) 'SOLUTION VECTOR X'
      WRITE(*,*)
      WRITE(*,*) (X(I), I = 1, N)
      WRITE(*,*)
      STOP
      END
* ------------ End of main program LEG1 ------------
* ---------------------------------------------------------- *
*     SUBROUTINE GAUSS2(N,A,B,X)
* ---------------------------------------------------------- *
* Subroutine                                                 *
*   This subroutine solves a system of linear                *
*   equations using Gauss elimination method with            *
*   partial pivoting                                         *
* ---------------------------------------------------------- *
* Arguments                                                    *
* Input                                                        *
*     A - Coefficient matrix                                   *
*     B - Right Side vector                                    *
*     N - Size of the system                                   *
* Output                                                       *
*     X - Solution vector                                      *
* ------------------------------------------------------------ *
* Local Variables                                              *
*   NIL                                                        *
* ------------------------------------------------------------ *
* Functions invoked                                            *
*   NIL                                                        *
* ------------------------------------------------------------ *
* Subroutines called                                           *
*   ELIM,BSUB                                                  *
* ------------------------------------------------------------ *
    REAL A,B,X
    INTEGER N
    EXTERNAL ELIM, BSUB
    DIMENSION A(10,10), B(10), X(10)
* Forward elimination
    CALL ELIM(N,A,B)
* Solution by back substitution
    CALL BSUB(N,A,B,X)
    RETURN
    END
* ------------- End of subroutine GAUSS2 ------------         *
* ------------------------------------------------------------ *
    SUBROUTINE ELIM(N,A,B)
* ------------------------------------------------------------ *
* Subroutine                                                   *
*   This subroutine performs forward elimination               *
*   incorporating partial pivoting technique                   *
* ------------------------------------------------------------ *
* Arguments                                                    *
* Input                                                        *
*     A - Coefficient matrix                                   *
*     B - Right side vector                                    *
*     N - System size                                          *
* Output                                                       *
*     A - Modified A                                           *
*     B - Modified B                                           *
* ------------------------------------------------------------ *
* Local Variables                                              *
*   FACTOR                                                     *
* ------------------------------------------------------------ *
* Functions invoked                                            *
*   NIL                                                        *
* ------------------------------------------------------------ *
  Subroutines called                                           *
*   PIVOT                                                      *
* ------------------------------------------------------------ *
    REAL A,B,X,FACTOR
    INTEGER N
    EXTERNAL PIVOT
    DIMENSION A(10,10),B(10)
    DO 33 K = 1, N-1
      CALL PIVOT (N,A,B,K)
      DO 22 I = K+1, N
         FACTOR = A(I,K)/A(K,K)
         DO 11 J = K+1, N
           A(I,J) = A(I,J) - FACTOR * A(K,J)
11       CONTINUE
         B(I) = B(I) - FACTOR * B(K)
22    CONTINUE
33  CONTINUE
    RETURN
    END
* -------------- End of subroutine ELIM ------------- *
* ------------------------------------------------------------ *
   SUBROUTINE PIVOT(N,A,B,K)
* ------------------------------------------------------------ *
* Subroutine                                                   *
*   This subroutine performs the task of partial               *
*   pivoting (reordering of equations)                         *
* ------------------------------------------------------------ *
* Arguments                                                    *
* Input                                                        *
*     N - System size                                          *
*     A - Coefficients matrix                                  *
*     B - Right side vector                                    *
*     K - Row under consideration for pivoting
* Output                                                       *
*     A - Modified A (after pivoting)                          *
*     B - Modified B (after pivoting)                          *
* ------------------------------------------------------------ *
* Local Variables                                              *
*   LARGE,TEMP,P                                               *
* ------------------------------------------------------------ *
* Functions invoked                                          *
*   ABS                                                      *
* ---------------------------------------------------------- *
* Subroutines called                                         *
*   NIL                                                      *
* ---------------------------------------------------------- *
    REAL LARGE,TEMP,A,B
    INTEGER P,N,K
    INTRINSIC ABS
    DIMENSION A(10,10), B(10)
* Find pivot P
    P = K
    LARGE = ABS(A(K,K))
    DO 11 I = K+1, N
      IF(ABS(A(I,K)) .GT. LARGE) THEN
            LARGE = ABS(A(I,K))
            P = I
      ENDIF
11  CONTINUE
* Exchange rows P and K
    IF(P.NE.K) THEN
      DO 22 J = K,N
            TEMP = A(P,J)
            A(P,J) = A(K,J)
            A(K,J) = TEMP
22    CONTINUE
      TEMP = B(P)
      B(P) = B(K)
      B(K) = TEMP
    ENDIF
    RETURN
    END
* --------------End of subroutine PIVOT------------------------ *
* ---------------------------------------------------------- *
    SUBROUTINE BSUB(N,A,B,X)
* ---------------------------------------------------------- *
* Subroutine                                                 *
*   This subroutine obtains the solution vector X            *
*   by back substitution                                     *
* ---------------------------------------------------------- *
* Arguments                                                  *
* Input                                                      *
*     N - System size                                        *
*     A - Coefficient matrix (after elimination)             *
*     B - Right side vector (after elimination)              *
* Output                                                     *
*     X  -  Solution  vector                                 *
* -----------------------------------------------------------*
* Local  Variables                                           *
    SUM                                                      *
  -----------------------------------------------------------*
* Functions  invoked                                         *
*   NIL                                                      *
* -----------------------------------------------------------*
* Subroutines  called                                        *
*   NIL                                                      *
* -----------------------------------------------------------*
    INTEGER  N
    REAL  A,B,X,  SUM
    DIMENSION  A(10,10),  B(10),  X(10)
    X(N)  =  B(N)/A(N,N)
    DO  55  K  =  N-1,  1,  -1
      SUM  =  0
      DO  44  J  =  K+1,  N
            SUM  =  SUM  +  A(K,J)  *  X(J)
44    CONTINUE
    X(K)  =  (B(K)  -  SUM)/A(K,K)
55  CONTINUE
    RETURN
    END
* ------------- End  of  subroutine  BSUB ------------       *
```

***Test Run Results***

```
            GAUSS  METHOD  WITH  PARTIAL  PIVOTING
What  is  the  size  of  the  system(n)?
3
Input  coefficients  a(i,j),  row-wise
one  row  on  each  line
2  2  1
4  2  3
1  1  1
Input  vector  b
6  4  0
SOLUTION  VECTOR  X
   5.0000000           1.0000000         -6.0000000
Stop  -  Program  terminated.
```

## 7.6 GAUSS-JORDAN METHOD

Gauss-Jordan method is another popular method used for solving a system of linear equations. Like Gauss elimination method, Gauss-Jordan method also uses the process of elimination of variables, but there is a major difference between them. In Gauss elimination method, a variable is eliminated from the rows below the pivot equation. But in Gauss-Jordan method, it is eliminated from all other rows (both below and above). This process thus eliminates all the off-diagonal terms producing a diagonal matrix rather than a triangular matrix. Further, all rows are normalised by dividing them by their pivot elements. This is illustrated in Fig. 7.4. Consequently, we can obtain the values of unknowns directly from the b vector, without employing back-substitution. Algorithm 7.3 enumerates the Gauss-Jordan elimination steps.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a'_{22} & a'_{23} \\ 0 & 0 & a''_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b'_2 \\ b''_3 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b'_1 \\ b''_2 \\ b'''_3 \end{bmatrix}$$

Result of Gauss elimination Result of Gauss-Jordan elimination

**Fig. 7.4** Comparison of Gauss and Gauss-Jordan methods of elimination

### Gauss-Jordan elimination

1. Normalise the first equation by dividing it by its pivot element.
2. Eliminate $x_1$ term from all the other equations.
3. Now, normalise the second equation by dividing it by its pivot element.
4. Eliminate $x_2$ from all the equations, above and below the normalised pivotal equation.
5. Repeat this process until $x_n$ is eliminated from all but the last equation.
6. The resultant $b$ vector is the solution vector.

**Algorithm 7.3**

The Gauss-Jordan method requires approximately 50 per cent more arithmetic operations compared to Gauss method. Therefore, this method is rarely used.

### Example 7.4

Solve the system

$$2x_1 + 4x_2 - 6x_3 = -8$$

$$x_1 + 3x_2 + x_3 = 10$$
$$2x_1 - 4x_2 - 2x_3 = -12$$

using Gauss-Jordan method.

*Step 1:* Normalise the first equation by dividing it by 2 (pivot element). The result is:

$$x_1 + 2x_2 - 3x_3 = -4$$
$$x_1 + 3x_2 + x_3 = 10$$
$$2x_1 - 4x_2 - 2x_3 = -12$$

*Step 2:* Eliminate $x_1$ from the second equation, subtracting 1 time the first equation from it. Similarly, eliminate $x_1$ from the third equation by subtracting 2 times the first equation from it. The result is:

$$x_1 + 2x_2 - 3x_3 = -4$$
$$0 + x_2 + 4x_3 = 14$$
$$0 - 8x_2 + 4x_3 = -4$$

*Step 3:* Normalise the second equation. (Note that it is already in normalised form.)

*Step 4:* Following similar approach, eliminate $x_2$ from first and third equations. This gives

$$x_1 + 0 - 11x_3 = -32$$
$$0 + x_2 + 4x_3 = 14$$
$$0 + 0 + 36x_3 = 108$$

*Step 5:* Normalise the third equation

$$x_1 + 0 - 11x_3 = -32$$
$$0 + x_2 + 4x_3 = 14$$
$$0 + 0 + x_3 = 3$$

*Step 6:* Eliminate $x_3$ from the first and second equations. We get

$$x_1 + 0 + 0 = 1$$
$$0 + x_2 + 0 = 2$$
$$0 + 0 + x_3 = 3$$

## Computational Effort

The Gauss-Jordan method requires only the elimination process. To eliminate $x_k$ from all but the $k$th equation, we need to undertake the following tasks:

1. Divide the coefficients $x_{k+1}, x_{k+2}, \dots x_n$ and $b_k$ by the coefficient of $x_k$.
2. Subtract suitable multiples of the $k$th equation from the other $(n - 1)$ equations to eliminate $x_k$ from these equations.

These tasks require:

| | |
|---|---|
| $(n - k + 1)$ | divisions |
| $(n - 1)(n - k + 1)$ | multiplications |
| $(n - 1)(n - k + 1)$ | subtractions |

Therefore, the total operations required in order to complete the elimination process are:

$$\text{Multiplications} = \sum_{k=1}^{n} (n-1)(n-k+1) = \frac{1}{2} n (n^2 - 1)$$

$$\text{Subtractions} = \sum_{k=1}^{n} (n-k+1) = \frac{1}{2} n (n^2 + 1)$$

$$\text{Divisions} = \sum_{k=1}^{n} (n-1)(n-k+1) = \frac{1}{2} n (n-1)$$

We see that the number of multiplications and subtractions is approximately equal to (1/2) $n^3$ and the number of divisions is (1/2) $n^2$. Computational efforts required by the Gauss and Gauss-Jordan methods are given in Table 7.2.

**Table 7.2** Comparison of computational effort

| | *Gauss method* | *Gauss-Jordan method* |
|---|---|---|
| Multiplication | $\frac{1}{3}n^3$ | $\frac{1}{2}n^3$ |
| Subtraction | $\frac{1}{3}n^3$ | $\frac{1}{2}n^3$ |
| Divisions | $\frac{1}{2}n^2$ | $\frac{1}{2}n^2$ |

It shows that the Gauss method requires only two-third of the number of multiplications or subtractions that the Gauss-Jordan method requires: i.e., the Gauss-Jordan method requires 50 per cent more multiplications and subtractions as pointed out earlier.

## 7.7 TRIANGULAR FACTORISATION METHODS

The coefficient matrix **A** of a system of linear equations can be factorised (or decomposed) into two triangular matrices **L** and **U** such that

$$\mathbf{A} = \mathbf{LU} \tag{7.9}$$

where

$$\mathbf{L} = \begin{bmatrix} l_{11} & 0 & \dots & 0 \\ l_{21} & l_{22} & \dots & 0 \\ \vdots & \vdots & & \vdots \\ l_{n1} & l_{n2} & \dots & l_{nn} \end{bmatrix}$$

and

$$\mathbf{U} = \begin{bmatrix} u_{11} & u_{12} & \dots & u_{1n} \\ 0 & u_{22} & \dots & u_{2n} \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \dots & u_{nn} \end{bmatrix}$$

L is known as lower triangular matrix and U is known as upper triangular matrix.

Once A is factorised into L and U, the system of equations

$$\mathbf{A}x = b$$

can be expressed as follows

$$(\mathbf{LU})\, x = b$$

or

$$\mathbf{L}\,(\mathbf{U}x) = b \tag{7.10}$$

Let us assume that

$$\boxed{\mathbf{U}x = z} \tag{7.11}$$

where $z$ is an unknown vector. Substituting Eq. (7.11) in equation (7.10), we get

$$\boxed{\mathbf{L}z = b} \tag{7.12}$$

Now we can solve the system

$$\mathbf{A}x = b$$

in two stages:

1. Solve the equation

   $$\mathbf{L}z = b$$

   for $z$ by forward substitution
2. Solve the equation

   $$U_x = z$$

   for $x$ using $z$ (found in stage 1) by back substitution.

The elements of L and U can be determined by comparing the elements of the product of L and U with those of A. The process produces a system of $n^2$ equations with $n^2 + n$ unknowns ($l_{ij}$ and $m_{ij}$) and, therefore, L and U are not unique. In order to produce unique factors, we should reduce the number of unknowns by $n$.

This is done by assuming the diagonal elements of L or U to be unity. The decomposition with L having unit diagonal values is called the *Dolittle LU decomposition* while the other one with U having unit diagonal elements is called the *Crout LU decomposition*.

## Dolittle Algorithm

We can solve for the components of L and U, given A, as follows:

$$\mathbf{A} = \mathbf{LU}$$

implies that

$$a_{ij} = l_{i1}\, u_{1j} + l_{i2}\, u_{2j} + \ldots + l_{ii}\, u_{ij} \qquad \text{for } i < j \tag{7.13}$$

$$a_{ij} = l_{i1}\, u_{1j} + l_{i2}\, u_{2j} + \ldots + l_{ii}\, u_{jj} \qquad \text{for } i = j \tag{7.14}$$

$$a_{ij} = l_{i1}\, u_{1j} + l_{i2}\, u_{2j} + \ldots + l_{ij}\, u_{jj} \qquad \text{for } i > j \tag{7.15}$$

where $u_{ij} = 0$ for $i > j$ and $l_{ij} = 0$ for $i < j$

The Dolittle algorithm assumes that all the diagonal elements of **L** are unity. That is

$$l_{ii} = 1, \qquad i = 1, 2, \ldots n.$$

Using equations (7.13), (7.14) and (7.15), we can successively determine the elements of **U** and **L** as follows:

If $i \le j$

$$\boxed{u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik}\, u_{kj} \qquad j = 1, 2, \ldots n}$$

where $u_{11} = a_{11}$, $u_{12} = a_{12}$, $u_{13} = a_{13}$

Similarly, if $i > j$

$$\boxed{l_{ij} = \frac{l}{u_{ij}} \times \left[ a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj} \right] \qquad j = 1, 2, \ldots i - 1}$$

where $l_{11} = l_{22} = l_{33} = 1$ and $l_{i1} = a_{i1}/u_{11}$ for $i = 2$ to $n$.

Note that, for computing any element, we need the values of elements in the previous columns as well as the values of elements in the column above that element, as illustrated in Fig. 7.5. This suggests that we should compute the elements, *column by column* from *left to right* within each column from *top to bottom*.

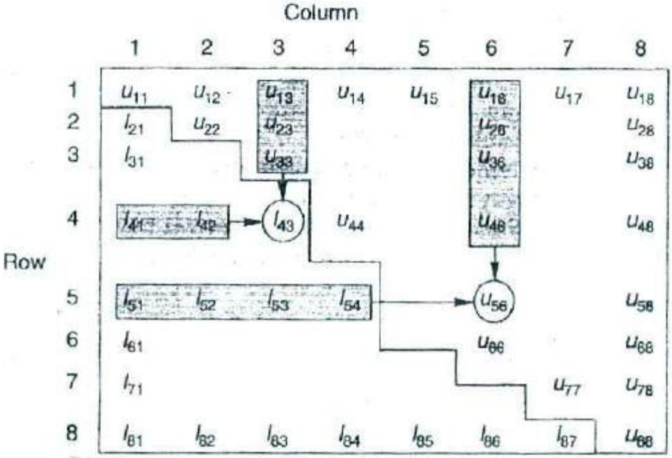


**Fig. 7.5** Pictorial view of Dolittle algorithm of LU decomposition

Algorithm 7.4 lists steps involved in LU decomposition and its application to the solution of linear equations.

*Note:*

1. There is no need to store 1's on the diagonal of **L** matrix.
2. There is also no need to store 0's of **L** or **U**. Consequently, the values of **L** can be stored in the zero space of **U**.
3. Further, each element of $a_{ij}$ is used only once (and never used again).

It is clear that we can "overwrite" **A** with **L** and **U** and save memory. This means the corresponding $l_{ij}$ or $u_{ij}$ can be stored in the location of $a_{ij}$.

### Dolittle LU decomposition and solution

```
1. Given $n$, $\mathbf{A}$, $b$
2. Set $u_{1j} = a_{1j}$            for $j = 1$ to $n$
   Set $l_{ii} = 1$                 for $i = 1$ to $n$
   Set $l_{i1} = a_{i1}/u_{11}$     for $i = 2$ to $n$
3. For each $j = 2$ to $n$ do:
     (i) For $i = 2$ to $j$
           Compute $u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}$
         Repeat $i$
     (ii) For $i = j + 1$ to $n$
           Compute $l_{ij} = \frac{1}{u_{jj}} \times \left[ a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \right]$
         Repeat $i$
   Repeat $j$
4. Set $z_1 = b_1$
5. For $i = 2$ to $n$
       Set sum $= \sum_{j=1}^{i-1} l_{ij} z_j$
       Set $z_i = b_i -$ sum
   Repeat $i$
6. Set $x_n = z_n / u_{nn}$
7. For $i = n - 1$ to 1
       Set sum$= \sum_{j=i+1}^{n} u_{ij} x_j$
       Set $x_i = (z_i -$ sum$) / u_{ii}$
   Repeat $i$
8. Write results
```

Algorithm 7.4

### Example 7.5

Solve the system

$$3x_1 + 2x_2 + x_3 = 10$$
$$2x_1 + 3x_2 + 2x_3 = 14$$
$$x_1 + 2x_2 + 3x_3 = 14$$

by using Dolittle LU decomposition method

*Factorisation*

For $i = 1$, $l_{11} = 1$ and

$$u_{11} = a_{11} = 3$$
$$u_{12} = a_{12} = 2$$
$$u_{13} = a_{13} = 1$$

For $i = 2$

$$l_{21} = \frac{a_{21}}{u_{11}} = \frac{2}{3} \qquad \text{and} \qquad l_{22} = 1$$

$$u_{22} = a_{22} - l_{21}\, u_{12} = 3 - \frac{2}{3} \times 2 = \frac{5}{3}$$

$$u_{23} = a_{23} - l_{21} u_{13} = 2 - \frac{2}{3} \times 1 = \frac{4}{3}$$

For $i = 3$

$$l_{31} = \frac{a_{31}}{u_{11}} = \frac{1}{3}$$

$$l_{32} = \frac{a_{32} - l_{31} u_{12}}{u_{22}}$$

$$= \frac{2 - 1/3 \times 2}{5/3} = \frac{4}{5}$$

$$l_{33} = 1$$

$$u_{33} = a_{33} - l_{31}\, u_{13} - l_{32}\, u_{23}$$

$$= 3\frac{1}{3} \times 1 - \frac{4}{5} \times \frac{4}{3} = \frac{24}{15}$$

Thus, we have

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ 2/3 & 1 & 0 \\ 1/3 & 4/5 & 1 \end{bmatrix}$$

$$\mathbf{U} = \begin{bmatrix} 3 & 2 & 1 \\ 0 & 5/3 & 4/3 \\ 0 & 0 & 24/15 \end{bmatrix}$$

*Forward Substitution*

Solving $\mathbf{L}\boldsymbol{z} = \boldsymbol{b}$ by forward substitution, we get

$$\begin{aligned}
z_1 &= b_1 = 10\\
z_2 &= b_2 - l_{21}\, z_1\\
&= 14 - 2/3 \times 10 = 22/3\\
z_3 &= b_3 - l_{31}\, z_1 - l_{32}\, z_2\\
&= 14 - 1/3 \times 10 - 4/5 \times 22/3 = \frac{72}{15}
\end{aligned}$$

*Back Substitution*

Solving $\mathbf{U}_x = \boldsymbol{z}$ by back substitution, we get

$$x_3 = \frac{27/15}{24/15} = 3$$

$$\begin{aligned}
x_2 &= \frac{z_2 - u_{23}x_3}{u_{22}}\\
&= \frac{(22/3) - (4/3)\times 3}{5/3} = 2
\end{aligned}$$

$$\begin{aligned}
x_1 &= \frac{z_1 - u_{12}x_2 - u_{13}x_3}{u_{11}}\\
&= \frac{10 - 2\times 2 - 1\times 3}{3} = 1
\end{aligned}$$

## Program DOLIT

The Dolittle LU decomposition method for solving a system of linear equations may be implemented on a computer using the program DOLIT. The DOLIT program solves a problem with the help of two subprograms, LUD and SOLVE.

The subprogram LUD decomposes the given coefficient matrix using the Dolittle algorithm and the resultant **L** and **U** matrices are supplied back to the main program DOLIT. Note that when it fails to decompose the matrix, a message to that effect is sent to the main program for necessary action.

The subprogram SOLVE receives the right side vector ***B*** and the decomposed matrices **L** and **U** from the main program and then obtains the solution vector ***X*** employing both the forward and backward substitution techniques.

```
* ---------------------------------------------------------- *
      PROGRAM DOLIT
* ---------------------------------------------------------- *
* Main program                                               *
*    This program solves a system of linear equations        *
*    using Dolittle LU decomposition                         *
* ---------------------------------------------------- *
* Functions invoked                                    *
*   NIL                                                *
* ---------------------------------------------------- *
* Subroutines used                                     *
*   LUD, SOLVE                                         *
* ---------------------------------------------------- *
* Variables used                                       *
*   N - System size                                    *
*   A - Coefficient matrix of the system               *
*   B - Right side vector                              *
*   L - Lower triangular matrix                        *
*   U - Upper triangular matrix                        *
*   FACT - Factorisation status                        *
* ---------------------------------------------------- *
* Constants used                                       *
*   YES,NO                                             *
* ---------------------------------------------------- *
      INTEGER N,YES,NO,FACT
      REAL A,U,L,B,X
      EXTERNAL LUD,SOLVE
      PARAMETER( YES = 1, NO = 0 )
      DIMENSION A(10,10),U(10,10),L(10,10),B(10),X(10)
      WRITE(*,*)
      WRITE(*,*) 'SOLUTION BY DOLITTLE METHOD '
      WRITE(*,*)
* Read input data
      WRITE(*,*) 'What is the size of A?'
      READ(*,*)   N
      WRITE(*,*) 'Input coefficients a(i,j), row-wise, ',
     +           'one row on each line'
      DO 10 I = 1, N
         READ(*,*) (A(I,J), J=1,N)
10    CONTINUE
      WRITE(*,*) 'Input vector B on one line'
      READ(*,*) (B(I), I=1, N)
* LU factorisation
      CALL LUD(N,A,U,L,FACT)
      IF( FACT .EQ. YES ) THEN
*        Print LU matrices
* Print matrix U
         WRITE(*,*)
         WRITE(*,*) 'MATRIX U'
         DO 20 I = 1,N
```

```
          WRITE(*,111) (U(I,J),J=1,N)
20      CONTINUE
* Print matrix L
        WRITE(*,*)
        WRITE(*,*) 'MATRIX L'
        DO 30 I=1,N
          WRITE(*,111) (L(I,J),J=1,N
30      CONTINUE
      ELSE
        WRITE(*,*)
        WRITE(*,*) 'FACTORISATION NOT POSSIBLE'
        WRITE(*,*)
        STOP
      ENDIF
* Solve for X
      CALL SOLVE(N,U,L,B,X)
      WRITE(*,*)
      WRITE(*,*) 'SOLUTION VECTOR X'
      WRITE(*,*)
      WRITE(*,111) (X(I), I=1,N)
      WRITE(*,*)
111 FORMAT(3F15.6)
      STOP
      END
* -------------End of main program DOLIT ------------ *
* ------------------------------------------------------ *
      SUBROUTINE LUD(N,A,U,L,FACT)
* ------------------------------------------------------ *
* Subroutine                                             *
*   This subroutine decomposes the matrix A into         *
*   L and U matrices using Dolittle algorithm            *
* ------------------------------------------------------ *
* Arguments                                              *
* Input                                                  *
*     N - System size                                    *
*     A - Coefficient matrix of the original system      *
* Output                                                 *
*     U - Decomposed upper triangular matrix             *
*     L - Decomposed lower triangular matrix             *
*     FACT - Fact about decomposition (yes or no)        *
* ------------------------------------------------------ *
* Local Variables                                        *
*   SUM                                                  *
* ------------------------------------------------------ *
* Functions invoked                                            *
*   NIL                                                        *
* ------------------------------------------------------------ *
* Subroutines called                                           *
*   NIL                                                        *
* ------------------------------------------------------------ *

      INTEGER N,YES,NO,FACT
      REAL A,U,L,SUM
      PARAMETER( YES = 1, NO = 0 )
      DIMENSION A(10,10), U(10,10), L(10,10)
* Initialise U and L matrices
      DO 1 I = 1,N
        DO 1 J = 1,N
          U(I,J) = 0.0
          L(I,J) = 0.0
1     CONTINUE
* Compute the elements of U and L
      DO 10 J = 1,N
        U(1,J) = A(1,J)
10    CONTINUE
      DO 20 I = 1,N
        L(I,I) = 1.0
20    CONTINUE
      DO 30 I = 2,N
        L(I,1) = A(I,1)/U(1,1)
30    CONTINUE
      DO 100 J = 2,N
        DO 50 I = 2,J
          SUM = A(I,J)
          DO 40 K = 1,I-1
            SUM = SUM - L(I,K) * U(K,J)
40        CONTINUE
            U(I,J) = SUM
50    CONTINUE
      IF( U(J,J) .LE. 1.E-6 ) THEN
        FACT = NO
        RETURN
      ENDIF
      DO 70 I = J+1,N
        SUM = A(I,J)
        DO 60 K = 1,J-1
          SUM = SUM - L(I,K) * U(K,J)
60    CONTINUE
```

```
      L(I,J) = SUM/U(J,J)
70    CONTINUE
100 CONTINUE
    FACT = YES
    RETURN
    END
* ---------------End of subroutine LUD-------------- *
* ------------------------------------------------- *
    SUBROUTINE SOLVE(N,U,L,B,X)
* ------------------------------------------------- *
* Subroutine                                         *
*   This subroutine obtains the solution vector X    *
*   using the coefficients of L and U matrices       *
* ------------------------------------------------- *
* Arguments                                          *
* Input                                              *
*   N - System size                                  *
*   U - Upper triangular matrix                      *
*   L - Lower triangular matrix                      *
*   B - Right side vector                            *
* Output                                             *
*   X - Solution vector                              *
* ------------------------------------------------- *
* Local Variables                                    *
*   SUM, Z(vector)                                   *
* ------------------------------------------------- *
* Functions invoked                                  *
*   NIL                                              *
* ------------------------------------------------- *
* Subroutines called                                 *
*   NIL                                              *
* ------------------------------------------------- *
    INTEGER N
    REAL U,L,SUM,B,X,Z
    DIMENSION U(10,10),L(10,10),B(10),X(10),Z(10)
* Forward substitution
    Z(1) = B(1)
    DO 20 I = 2,N
      SUM = 0.0
      DO 10 J = 1,I-1
        SUM = SUM + L(I,J) * Z(J)
10    CONTINUE
    Z(I) = B(I) - SUM
20  CONTINUE
* Back substitution
    X(N) = Z(N)/U(N,N)
```

```
        DO 40 I = N-1,1,-1
           SUM = 0.0
           DO 30 J = I+1,N
              SUM = SUM + U(I,J) * X(J)
30         CONTINUE
           X(I) = (Z(I) - SUM)/U(I,I)
40    CONTINUE
      RETURN
      END
* --------------End of subroutine SOLVE ------------- *
```

*Test Run Results*

```
SOLUTION BY DOLITTLE METHOD
What is the size of A?
3
Input coefficients a(i,j), row-wise, one row on each line
3 2 1
2 3 2
1 2 3
Input vector B on one line
10 14 14
MATRIX U
        3.000000          2.000000          1.000000
         .000000          1.666667          1.333333
         .000000           .000000          1.600000
MATRIX L
        1.000000           .000000           .000000
         .666667          1.000000           .000000
         .333333           .800000          1.000000
SOLUTION VECTOR X
        1.000000          2.000000          3.000000
Stop - Program terminated.
```

## Crout Algorithm

Another approach to LU decomposition is *Crout algorithm*. As mentioned earlier, Crout decomposition algorithm assumes unit diagonal values for **U** matrix and the diagonal elements of **L** matrix may assume any values as shown below.

$$\begin{bmatrix} l_{11} & 0 & \dots & 0 \\ l_{21} & l_{22} & \dots & 0 \\ \vdots & \vdots & & \vdots \\ l_{n1} & l_{n2} & \dots & l_{nn} \end{bmatrix} \begin{bmatrix} l & u_{12} & \dots & u_{1n} \\ 0 & 1 & \dots & u_{2n} \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \dots & l \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix}$$

We can use an approach that is similar to the one used in Dolittle decomposition to evaluate the elements of **L** and **U**.

## Cholesky Method

In case **A** is symmetric, the LU decomposition can be modified so that the upper factor is the transpose of the lower one (or vice versa). That is, we can factorise **A** as

$$\mathbf{A} = \mathbf{L}\mathbf{L}^{\mathrm{T}}$$

or

$$\mathbf{A} = \mathbf{U}^{\mathrm{T}}\,\mathbf{U} \tag{7.16}$$

Just as for Dolittle decomposition, by multiplying the terms of Eq. (7.16) and setting them equal to each other (see Eqs (7.13), (7.14) and (7.15)), the following recurrence relations can be obtained.

$$\boxed{\begin{aligned} u_{ij} &= \sqrt{a_{ii} - \sum_{k=1}^{i-1} u_{ki}^2} && (i = 1 \text{ to } n) \\ u_{ij} &= \frac{1}{u_{ii}}\left[a_{ij} - \sum_{k=1}^{i-1} u_{ki}\,u_{kj}\right] && (j > i) \end{aligned}} \tag{7.17}$$

This decomposition is called the *Cholesky's factorisation* or the *method of square roots*. Algorithm 7.5 lists the basic steps for computing the elements U, column by column.

**Cholesky's factorisation**

```
1. Given $n$, A
2. Set $u_{11} = \sqrt{a_{11}}$
3. Set $u_{1j} = a_{1j} / u_{11}$        for $i = 2$ to $n$
4. For $j = 2$ to $n$
       For $i = 2$ to $j$
              sum = $a_{ij}$
              For $k = 1$ to $i - 1$
                     sum = sum − $u_{ki}\,u_{kj}$
              Repeat $k$
              set $u_{ij}$ = sum / $u_{ii}$        if $i < j$
              set $u_{ij} = \sqrt{\text{sum}}$        if $i = j$
       Repeat $i$
   Repeat $j$
5. End of factorisation
```

**Algorithm 7.5**

**Example 7.6**

Factorise the matrix

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 8 & 22 \\ 3 & 22 & 82 \end{bmatrix}$$

using Cholesky's algorithm

For $i = 1$, according Eq. (7.17)

$$u_{11} = \sqrt{1} = 1$$

$$u_{12} = \frac{a_{12}}{u_{11}} = \frac{2}{1} = 2$$

$$u_{13} = \frac{a_{13}}{u_{11}} = \frac{3}{1} = 3$$

For $i = 2$

$$u_{22} = \sqrt{a_{22} - u_{12}^2} = \sqrt{8-4} = 2$$

$$u_{23} = \frac{a_{23} - u_{12}u_{13}}{u_{22}} = \frac{22 - 2\times 3}{2} = \frac{16}{2} = 8$$

For $i = 3$

$$u_{33} = \sqrt{a_{33} - u_{13}^2 - u_{23}^2}$$

$$= \sqrt{82 - 9 - 64} = \sqrt{9} = 3$$

Thus, we have

$$\mathrm{U} = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 2 & 8 \\ 0 & 0 & 3 \end{bmatrix}$$

## 7.8 ROUNDOFF ERRORS AND REFINEMENT

In all the direct methods, only one estimate of $x_i$ is produced. As we know, methods use a large number of floating point operations and, therefore, introduce roundoff errors in the final solution. We have no indication how accurate the solution is.

One way to check this is to substitute the answer back into the original equations to see whether a substantial error has occurred. In case the error is beyond the acceptable limit, the solution can be improved by a technique known as *iterative refinement*.

Let us suppose $x^{(1)}$ is the solution of the system

$$Ax = b$$

Substituting $x^{(1)}$ back in the original equation, we get

$$\mathbf{A}x^{(1)} = b'$$

Since $x^{(1)}$ is not exact, $b'$ is not equal to $b$. If we define

$$r^{(1)} = b^1 - b$$

then we have

$$r^{(1)} = \mathbf{A}x^{(1)} - b \tag{7.18}$$

where $r$ is known as *residual vector*. If we can use this information to compute the error, then we can correct the approximate solution with this error.

If we assume that $x^*$ is the exact solution and $e$ is the error in $x$, then

$$x^* = x^{(1)} - e^{(1)}$$

or

$$x^{(1)} = x^* + e^{(1)} \tag{7.19}$$

Substituting this in Eq. (7.18), we get

$$\begin{aligned} r^{(1)} &= \mathbf{A}\,(x^* + e^{(1)}) - b \\ &= \mathbf{A}x^* + \mathbf{A}e^{(1)} - b \end{aligned}$$

Since $\mathbf{A}x^* = b$, this results in

$$\boxed{\mathbf{A}e^{(1)} = r^{(1)}} \tag{7.20}$$

We can now obtain $e^{(1)}$ by solving Eq. (7.20) and then estimate the next improved solution as

$$x^{(2)} = x^{(1)} - e^{(1)}$$

If we need further improvement, we can repeat the process by calculating $e^{(2)}$ using

$$\mathbf{A}e^{(2)} = r^{(2)}$$

where

$$r^{(2)} = \mathbf{A}x^{(2)} - b$$

We get the next estimate as

$$x^{(3)} = x^{(2)} - e^{(2)}$$

This process can be repeated as many times as we wish to achieve a desired accuracy. Algorithm 7.6 lists the steps for implementing the iterative refinement process.

### Iterative refinement

1. Obtain LU factorisation of **A**
2. Compute the solution $x$ by forward and back substitutions
3. Find the residual vector $r$ using
   $$r = \mathbf{A}x - b$$
4. Compute the error using
   $$Ae = r$$
   by forward and back substitutions

*(Contd.)*

*(contd.)*

```
5. Set $\mathbf{x} = \mathbf{x} - e$
6. If $e$ is sufficiently small
        stop
   otherwise
        go to step 3
```

Algorithm 7.6

## 7.9 ILL-CONDITIONED SYSTEMS

As pointed out in the beginning of the chapter, arriving at a proper solution depends on the condition of the system. Systems where small changes in the coefficient result in large deviations in the solution are said to be *ill-conditioned systems*. A wide range of answers can satisfy such equations. This means that a completely erroneous set of answers may produce zero (or near zero) residuals. This is illustrated in Example 7.7.

Ill-conditioned systems are very sensitive to roundoff errors. These errors during computing process may induce small changes in the coefficients which, in turn, may result in a large error in the solution.

We can decide the condition of a system either graphically or mathematically. Graphically, if two lines appear almost parallel, then we can say the system is ill-conditioned, since it is hard to decide just at which point they intersect.

The problem of ill-condition can be mathematically described as follows: consider a two equation system

$$a_{11}\, x_1 + a_{12}\, x_2 = b_1$$

$$a_{21}\, x_1 + a_{22}\, x_2 = b_2$$

If these two lines are almost parallel, their slopes must by nearly equal. That is

$$\boxed{\frac{a_{11}}{a_{12}} \approx \frac{a_{21}}{a_{22}}}$$

Alternatively,

$$a_{11}a_{22} \approx a_{12}a_{21}$$

or

$$\boxed{a_{11}a_{22} - a_{12}a_{21} \approx 0}$$

Note that $a_{11}\, a_{22} - a_{12}\, a_{21}$ is the determinant of the coefficient matrix

$$\mathrm{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

This shows that the determinant of an ill-conditioned system is very small or nearly equal to zero.

In partial pivoting technique, we try to interchange the rows so that the largest element becomes the pivot element. This is done basically to avoid a division by zero or nearly zero point. Even the largest element in that column may happen to be zero (or nearly zero). Such situations arise when the systems are ill-conditioned. Solution of these systems may not be meaningful.

### Example 7.7

Solve the following equations

$$2x_1 + x_2 = 25$$

$$2.001x_1 + x_2 = 25.01$$

and thereby discuss the effect of ill-conditioning.

$$x_1 = \frac{25 \times 1 - 25.01 \times 1}{2 \times 1 - 2.001 \times 1} = 10$$

$$x_2 = \frac{25.01 \times 2 - 25 \times 2.001}{2 \times 1 - 2.001 \times 1} = 5$$

Let us change the coefficient of $x_1$ in the second equation to 2.0005. Now the values of $x_1$ and $x_2$ are

$$x_1 = \frac{25 - 25.01}{2 - 2.0005} = 20$$

$$x_2 = \frac{25.01 \times 2 - 25 \times 2.0005}{2 - 2.0005} = -15$$

Compare the results. A small change in one of the coefficients has resulted in a large change in the result.

If we substitute these values back into the equations, we get the residuals as

$$r_1 = 40 - 15 - 25 \qquad = 0$$

$$r_2 = 40.02 - 15 - 25.01 = 0.01$$

The first equation is satisfied exactly and the residual of the second is small. It appears as if the results are correct. This illustrates the effect of roundoff errors on ill-conditioned systems.

## 7.10 MATRIX INVERSION METHOD

Another way to obtain the solution of an equation of type

$$\mathbf{A}x = b \tag{7.21}$$

is by using matrix algebra. Multiply each side of Eq. (7.21) by the inverse of **A**. This yields

$$\mathbf{A}^{-1}\,\mathbf{A}x = \mathbf{A}^{-1}\,b \tag{7.22}$$

since $\mathbf{A}^{-1}\,\mathbf{A} = \mathbf{I}$, the identity matrix, Equation (7.22) becomes

$$\boxed{x = \mathbf{A}^{-1}\,b} \tag{7.23}$$

Equation (7.23) gives the solution for $x$.

This approach becomes useful when we need to solve Eq. (7.21) for different sets of $b$ values while **A** remains the same.

## Computing Matrix Inverse

Although the Gauss-Jordan method is more complicated compared to Gauss elimination method, this method provides a simple approach for obtaining the inverse of a matrix.

This is done as follows:

1. Augment the coefficient matrix **A** with an identity matrix as shown below:

$$\left[\begin{array}{ccc|ccc} a_{11} & a_{12} & a_{13} & 1 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 1 & 0 \\ a_{31} & a_{32} & a_{33} & 0 & 0 & 1 \end{array}\right]$$

2. Apply the Gauss-Jordan method to the augmented matrix to reduce **A** to an identity matrix. The result will be as shown below:

$$\left[\begin{array}{ccc|ccc} 1 & 0 & 0 & a'_{11} & a'_{12} & a'_{13} \\ 0 & 1 & 0 & a'_{21} & a'_{22} & a'_{23} \\ 0 & 0 & 1 & a'_{31} & a'_{32} & a'_{33} \end{array}\right]$$

The right-hand side of the augmented matrix is the inverse of **A**. Now, we can obtain the solution as follows:

$$x_1 = a'_{11} \times b_1 + a'_{12} \times b_2 + a'_{13} \times b_3$$

$$x_2 = a'_{21} \times b_1 + a'_{22} \times b_2 + a'_{23} \times b_3$$

$$x_3 = a'_{31} \times b_1 + a'_{32} \times b_2 + a'_{33} \times b_3$$

## Condition Number

The inverse matrix can also be used to decide whether a system is ill-conditioned. Let us define a matrix **C** as

$$\mathbf{C} = \mathbf{A}\,.\,\mathbf{A}^{-1} \tag{7.24}$$

If **C** is close to identity matrix, then the system is well-conditioned. If not, it indicates ill-conditioning.

Equation (7.24) can be expressed using the concept of matrix norm as follows:

$$\text{cond}\,(\mathbf{A}) = ||\mathbf{A}|| \cdot ||\mathbf{A}^{-1}|| \tag{7.25}$$

where cond(**A**) is called the *condition number* and || **A** || is the "norm" of the matrix **A**. The norm is defined as follows

$$||\mathbf{A}|| = \max_{1 \leq i \geq n} \sum_{j=1}^{n} |a_{ij}|$$

This is known as *row-sum norm*. In this norm, the sum of the absolute values of the elements for each row is computed and the largest of these is taken as the norm.

The smaller the condition number, the better is matrix **A** suited to numerical computation.

## 7.11 SUMMARY

In this chapter we studied systems of linear equations. Among the two popular approaches available for solving these equations, we considered the elimination (also known as direct) methods in detail. They include:

- Gauss elimination method (basic)
- Gauss elimination with pivoting
- Gauss-Jordan method
- LU decomposition method using Dolittle algorithm
- Matrix inverse method

We also stated that other LU decomposition techniques, such as Crout algorithm and Cholesky's factorisation, may be applied to solve the equations.

Direct methods introduce roundoff errors. We presented an iterative refinement procedure for improving the final result.

Computer programs with test results have been given for the following methods:

- Basic Gauss elimination method
- Gauss elimination with partial pivoting
- Dolittle LU decomposition method

### Key Terms

*Back substitution*
*Basic Gauss' elimination*
*Cholesky's algorithm*
*Cholesky's factorisation*
*Complete pivoting*
*Condition number*
*Crout algorithm*
*Crout LU decomposition*
*Lower triangular matrix*
*LU decomposition*
*Matrix inversion*
*Matrix norm*
*Method of square roots*
*Modular structure*
*Nonlinear*
*Normalisation*

*(Contd.)*

*(Contd.)*

*Decomposition*
*Dependent equations*
*Direct method*
*Dolittle LU decomposition*
*Elimination approach*
*Forward elimination*
*Gauss elimination*
*Gauss-Jordan method*
*Homogeneous equations*
*Ill-conditioned system*
*Inconsistent equations*
*Infinite solutions*
*Iterative refinement*
*Linear*
*Over-determined*
*Partial pivoting*
*Pivot element*
*Pivot equation*
*Pivoting*
*Residual vector*
*Row-sum norm*
*Simultaneous equations*
*Singular systems*
*Triangularisation*
*Under-determined*
*Unique solution*
*Upper triangular matrix*
*Zero residuals*

## REVIEW QUESTIONS

1. Describe the two basic approaches that are employed for solving a system of linear equations.
2. What are the four possible solution conditions of a system of linear equations? Explain each one of them with an illustration.
3. Explain under-determined and over-determined systems.
4. What is meant by homogenous equations?
5. State some basic rules that are used in the elimination method of solving simultaneous linear equations.
6. Explain the basic concepts used in the Gauss elimination approach.
7. What is triangularisation of equations? How does it help obtain the solution?
8. What is pivoting? Distinguish between partial pivoting and complete pivoting.
9. How does pivoting improve accuracy of solution?
10. Compare critically Gauss elimination and Gauss-Jordan methods of solving simultaneous equations.
11. Show that Gauss-Jordan takes about 50% more operations than Gauss elimination for the case of three equations.
12. What is Dolittle decomposition? How is it different from Crout decomposition?
13. What is Cholesky's factorisation?
14. What is iterative refinement? How is it used to improve the accuracy of results?
15. What is meant by ill-conditioned systems?
16. Can we solve an ill-conditioned system? If yes, how?
17. What is condition number of a system? How is it computed?

## REVIEW EXERCISES

$x_1 =$
$x_2 =$

1. Solve the following system of equations using simple elimination process: NWB

$$\begin{aligned} x + y + z &= 6 \\ 2x - y + 3z &= 4 \\ 4x + 5y - 10z &= 13 \end{aligned}$$

2. Show that the following system of equations has no solution.

$$\begin{aligned} -2x + y + 3z &= 12 \\ x + 2y + 5z &= 4 \\ 6x - 3y - 9z &= 24 \end{aligned}$$

If $a_1 b_2 c_3 = a_3 b_2 c_1$ Then has no solution

3. Show that the following system of equations has infinite number of solutions.

$$\begin{aligned} x + y + z &= 20 \\ 2x - 3y + z &= -5 \\ 3x - 2y + 2z &= 15 \end{aligned}$$

✓

4. Solve the following systems of equations by simple Gauss elimination.

NWB (a)
$$\begin{aligned} 2x_1 + 3x_2 + 4x_3 &= 5 \\ 3x_1 + 4x_2 + 5x_3 &= 6 \\ 4x_1 + 5x_2 + 6x_3 &= 7 \end{aligned}$$

(b)
$$\begin{aligned} 2x_1 + 3x_2 + 4x_3 &= 5 \\ 3x_1 + 4.5x_2 + 5x_3 &= 6 \\ 4x_1 + 5x_2 + 6x_3 &= 7 \end{aligned}$$

(c)
$$\begin{aligned} x_1 + 2x_2 + 3x_3 &= 8 \\ 2x_1 + 4x_2 + 9x_3 &= 8 \\ 4x_1 + 3x_2 + 2x_3 &= 2 \end{aligned}$$

✓

5. Solve the systems in Exercise 4 using partial pivoting.
6. Solve the systems in Exercise 4 using complete pivoting.
7. Using Gauss elimination with partial pivoting, solve the following sets of equations.

(a)
$$\begin{aligned} 2x_1 + x_2 + x_3 - 2x_4 &= 0 \\ 4x_1 \qquad + 2x_3 + x_4 &= 8 \\ 3x_1 + 2x_2 + 2x_3 &= 7 \\ x_1 + 3x_2 + 2x_3 &= 3 \end{aligned}$$

(b)
$$\begin{aligned} x_1 + x_2 - 2x_3 &= 3 \\ 4x_1 - 2x_2 + x_3 &= 5 \\ 3x_1 - x_2 + 3x_3 &= 8 \end{aligned}$$

8. Solve the following systems of equations by Gauss-Jordan method

(a) 
$$\begin{aligned} x_1 + 2x_2 - 3x_3 &= 4 \\ 2x_1 + 4x_2 - 6x_3 &= 8 \\ x_1 - 2x_2 + 5x_3 &= 4 \end{aligned}$$

(b) 
$$\begin{aligned} 2x_1 + x_2 + x_3 &= 7 \\ 4x_1 + 2x_2 + 3x_3 &= 4 \\ x_1 - x_2 + x_3 &= 0 \end{aligned}$$

9. Find the Dolittle LU decompositions of the coefficient matrices of the systems in Exercises 7 and 8.

10. Solve the systems in Exercises 7 and 8 using the matrices L and U found in exercise 9 by forward and backward substitutions.

11. Find the Cholesky decomposition of the matrix

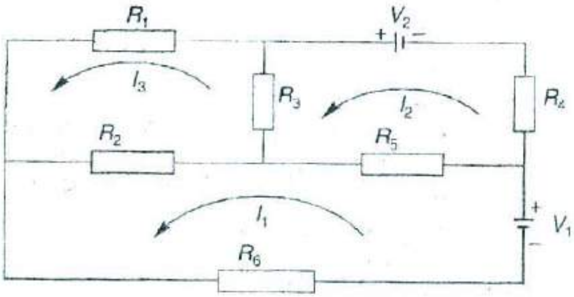
$$\begin{bmatrix} 4 & 1 & 1 \\ 1 & 5 & 2 \\ 1 & 2 & 3 \end{bmatrix}$$

12. Find the inverse of the following matrices using Gauss-Jordan elimination technique

(a) $\begin{bmatrix} 2 & 3 & 4 \\ 4 & 2 & 3 \\ 3 & 4 & 2 \end{bmatrix}$ (b) $\begin{bmatrix} 1 & 2 & -3 \\ 2 & 4 & -6 \\ -1 & -2 & 3 \end{bmatrix}$

13. Find the condition numbers of the coefficient matrices of systems in Exercise 4.

14. Consider the following electrical network connecting six resistors and two batteries:



Ohm's law states that the voltage across a resistor equals the current through it multiplied by its resistance. Using this law, we can set up the following equations:

$$\begin{aligned} R_6 I_1 + R_5(I_1 - I_2) + R_2(I_1 - I_3) &= V_1 \\ R_4 I_2 + R_3(I_2 - I_3) + R_5(I_2 - I_1) &= V_2 \\ R_1 I_3 + R_2(I_3 - I_1) + R_3(I_3 - I_2) &= 0 \end{aligned}$$

Assuming $R_1 = R_2 = R_3 = 2$, $R_4 = R_5 = R_6 = 3$ and $V_1 = V_2 = 5$, Solve the system of equations for currents $I_1$, $I_2$ and $I_3$ using Gauss elimination or Gauss-Jordan method.

15. A company produces four different products. They are processed through four different departments $A$, $B$, $C$ and $D$. The table below gives the number of hours that each department spends on each product.

| *Department* | *Products* | | | |
|---|---|---|---|---|
| | $P_1$ | $P_2$ | $P_3$ | $P_4$ |
| D1 | 2 | 3 | 1 | 2 |
| D2 | 1 | 2 | 2 | 4 |
| D3 | 3 | 4 | 4 | 5 |
| D4 | 3 | 2 | 2 | 3 |

Total production hours available each month in each department is as follows:

| Department | D1 | D2 | D3 | D4 |
|---|---|---|---|---|
| Hours | 265 | 260 | 352 | 250 |

Formulate the appropriate system of linear equations to determine the quantities of the four products that can be produced in each month, so that all the hours available in all departments are fully utilised. Determine how much time each department spends for each product.

## PROGRAMMING PROJECTS

1. Program LEG2 solves a system of linear equations using Gauss elimination with partial pivoting. Modify the program to implement complete pivoting.
2. Develop a program to factorise a matrix using Cholesky's algorithm.
3. Design and develop a program to implement the Gauss-Jordan elimination method for solving a system of linear equations.
4. Write a program to implement the Crout decomposition solution of linear equations.
5. Construct a program to implement the iterative refinement process as given in Algorithm 7.6.

CHAPTER 8

# Iterative Solution of Linear Equations

## 8.1 NEED AND SCOPE

Direct methods discussed in the previous chapter pose some problems when the systems grow larger or when most of the coefficients are zero. They require prohibitively large number of floating point operations and, therefore, not only become time consuming but also severely affect the accuracy of the solution due to roundoff errors. In such cases, iterative methods provide an alternative. For instance, ill-conditioned systems can be solved by iterative methods without facing the problem of roundoff errors.

The following three iterative methods are discussed in this chapter:

1. Jacobi iteration method
2. Gauss-Seidel iteration method
3. Successive over relaxation method

Like all other iterative processes, these methods introduce truncation errors and, therefore, it is important to understand the magnitude of this error as well as the rate of convergence of the iteration process.

## 8.2 JACOBI ITERATION METHOD

*Jacobi method* is one of the simple iterative methods. The basic idea behind this method is essentially the same as that for the fixed point method discussed in Chapter 6. Recall that an equation of the form

$$f(x) = 0$$

can be rearranged into a form

$$x = g(x)$$

The function $g(x)$ can be evaluated iteratively using an initial approximation $x$ as follows:

$$x_{i+1} = g(x_i) \qquad \text{for } i = 0, 1, 2...$$

Jacobi method extends this idea to a system of equations. It is a direct substitution method where the values of unknowns are improved by substituting directly the previous values.

Let us consider a system of $n$ equations in $n$ unknowns.

$$\begin{array}{c}
a_{11}\,x_1 + a_{12}\,x_2 + \dots + a_{1n}\,x_n = b_1 \\
a_{21}\,x_1 + a_{22}\,x_2 + \dots + a_{2n}\,x_n = b_2 \\
\cdot \quad \cdot \quad \cdot \quad \cdot \\
\cdot \quad \cdot \quad \cdot \quad \cdot \\
\cdot \quad \cdot \quad \cdot \quad \cdot \\
a_{n1}\,x_1 + a_{n2}\,x_2 + \dots + a_{nn}\,x_n = b_n
\end{array} \qquad (8.1)$$

Xamp

We rewrite the original system as

$$\begin{array}{l}
x_1 = \dfrac{b_1 - (a_{12}x_2 + a_{13}x_3 + \dots + a_{1n}x_n)}{a_{11}} \\[2ex]
x_2 = \dfrac{b_2 - (a_{21}x_1 + a_{23}x_3 + \dots + a_{2n}x_n)}{a_{22}} \\
\cdot \quad \cdot \quad \cdot \quad \cdot \\
\cdot \quad \cdot \quad \cdot \quad \cdot \\
\cdot \quad \cdot \quad \cdot \quad \cdot \\
x_n = \dfrac{b_n - (a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn-1}x_n)}{a_{nn}}
\end{array} \qquad (8.2)$$

Now, we can compute $x_1, x_2, \dots x_n$ by using initial guesses for these values. These new values are again used to compute the next set of $x$ values. The process can continue till we obtain a desired level of accuracy in the $x$ values.

In general, an iteration for $x_i$ can be obtained from the $i$th equation as follows

$$x_i^{(k+1)} = \frac{b_i - \left(a_{i1}x_1^{(k)} + a_{ii-1}x_{i-1}^{(k)} + a_{ii+1}x_{i+1}^{(k)} + \dots a_{in}x_n^{(k)}\right)}{a_{ii}} \qquad (8.3)$$

The computational steps of Jacobi iteration process are given in Algorithm 8.1.

### Jacobi iteration method

```
1. Obtain $n$, $a_{ij}$ and $b_i$ values.
2. Set $x_{0i} = b_i/a_{ii}$ for $i = 1,\ldots n$
3. Set key = 0
4. For $i = 1,2, \ldots n$
     (i) Set sum = $b_i$
    (ii) For $j = 1,2\ldots n$ ($j \neq i$)
             Set sum = sum $- a_{ij} x_{0j}$
         Repeat $j$
   (iii) Set $x_i$ = sum$/a_{ii}$
    (iv) if key = 0 then
         if $\left|\frac{x_i - x_{0i}}{x_i}\right| >$ error then
         set key = 1
   Repeat $i$
5. If key = 1 then
      set $x_{0i} = x_i$
      go to step 3
6. Write results
```

Algorithm 8.1

**Example 8.1**

Obtain the solution of the following system using the Jacobi iteration method

$$\begin{aligned} 2x_1 + x_2 + x_3 &= 5 \\ 3x_1 + 5x_2 + 2x_3 &= 15 \\ 2x_1 + x_2 + 4x_3 &= 8 \end{aligned}$$

First, solve the equations for unknowns on the diagonal. That is

$$x_1 = \frac{5 - x_2 - x_3}{2}$$

$$x_2 = \frac{15 - 3x_1 - 2x_3}{5}$$

$$x_3 = \frac{8 - 2x_1 - x_2}{4}$$

If we assume the initial values of $x_1$, $x_2$ and $x_3$ to be zero, then we get

$$x_1^{(1)} = \frac{5}{2} = 2.5$$

$$x_2^{(1)} = \frac{15}{5} = 3$$

$$x_3^{(1)} = \frac{8}{4} = 2$$

(Note that these values are nothing but$x_i^1 = b_i/a_{ii}$)

For the second iteration, we have

$$x_1^{(2)} = \frac{5-3-2}{2} = 0$$

$$x_2^{(2)} = \frac{15-3\times 2.5-2\times 2}{5} = \frac{3.5}{5} = 0.7$$

$$x_3^{(2)} = \frac{8-2\times 2.5-3}{4} = 0$$

After third iteration,

$$x_1^{(3)} = \frac{5-0.7}{2} = 2.15$$

$$x_2^{(3)} = \frac{15-3\times 0-2\times 0}{5} = 3$$

$$x_3^{(3)} = \frac{8-2\times 0-0.7}{4} = 1.825$$

After fourth iteration,

$$x_1^{(4)} = \frac{5-3-1.825}{2} = 0.0875$$

$$x_2^{(4)} = \frac{15-3\times 2.15-2\times 1.825}{4} = 1.225$$

$$x_3^{(4)} = \frac{8-2\times 2.15-3}{4} = 0.175$$

The process can be continued till the values of $x$ reach a desired level of accuracy.

## Program JACIT

The program JACIT solves a system of $n$ linear equations using the Jacobi iteration method as detailed in Algorithm 8.1. The main program reads interactively the system specifications and displays the results on the screen. The solution algorithm is implemented through the subroutine JACOBI.

The subprogram JACOBI, while computing the solution vector **X**, tests for the accuracy as well as the convergence. The computing process stops either when the desired accuracy is achieved or when the process does not converge within a specified number of iterations.

```
* -------------------------------------------------------- *
      PROGRAM JACIT
* -------------------------------------------------------- *
* Main program                                             *
*   This program uses the subprogram JACOBI to solve       *
*   a system of equations by Jacobi iteration method       *
* -------------------------------------------------------- *
* Functions invoked                                        *
*      NIL                                                 *
* -------------------------------------------------------- *
* Subroutines used                                         *
*      JACOBI                                              *
* -------------------------------------------------------- *
* Variables used                                           *
*      A - Coefficient matrix                              *
*      B - Right side vector                               *
*      N - System size                                     *
*      X - Solution vector                                 *
*      COUNT - Number of iterations completed              *
*      STATUS - Convergence status                         *
* -------------------------------------------------------- *
* Constants used                                           *
*      EPS - Error bound                                   *
*      MAXIT - Maximum iterations permitted                *
* -------------------------------------------------------- *
      REAL A,B,X,EPS
      INTEGER N,COUNT,MAXIT,STATUS
      PARAMETER(EPS=0.000001,MAXIT = 50)
      DIMENSION A(10,10), B(10), X(10)

      WRITE(*,*)
      WRITE(*,*)     'SOLUTION BY JACOBI ITERATION'
      WRITE(*,*)
      WRITE(*,*) 'What is the size of the system(n)?'
      READ(*,*) N
      WRITE(*,*) 'Input coefficients a(i,j), row-wise',
      WRITE(*,*) 'one row on each line'
      DO 20 I = 9, N
          READ(*,*) (A(I,J),J=1,N)
20    CONTINUE
      WRITE(*,*) 'Input vector b'
```

```
    READ(*,*) (B(I), I = 1, N)
    CALL JACOBI(N,A,B,X,EPS,COUNT,MAXIT,STATUS)
    IF(STATUS .EQ.2) THEN
      WRITE(*,*)
      WRITE(*,*) 'NO CONVERGENCE IN', MAXIT,
                 'ITERATIONS'
      WRITE(*,*)
    ELSE
      WRITE(*,*)
      WRITE(*,*) 'SOLUTION VECTOR X'
      WRITE(*,*)
      WRITE(*,*) (X(I), I = 1, N)
      WRITE(*,*)
      WRITE(*,*) 'ITERATIONS = ',COUNT
      WRITE(*,*)
    ENDIF
    STOP
    END
* ---------- End of main program JACIT -------------- *
* ---------------------------------------------------- *
  SUBROUTINE JACOBI(N,A,B,X,EPS,COUNT,MAXIT,STATUS)
* ---------------------------------------------------- *
* Subroutine                                           *
*    This subroutine solves a system of n linear       *
*    equations using the Jacobi iteration method       *
* ---------------------------------------------------- *
* Arguments                                            *
* Input                                                *
*    N - Number of equations                           *
*    A - Matrix of coefficients of the equations       *
*    B - Right side vector                             *
*    EPS - Error bound                                 *
*    MAXIT - Maximum iterations allowed                *
* Output                                               *
*    X - Solution vector                               *
*    COUNT - Number of iterations done                 *
*    STATUS - Convergence status                       *
* ---------------------------------------------------- *
* Local Variables                                      *
*    XO, SUM                                           *
* ---------------------------------------------------- *
* Functions invoked                                    *
*    ABS                                               *
* ---------------------------------------------------- *
* Subroutines called                                   *
*    NIL                                               *
* -----------------------------------------------------------------
      INTEGER  N,KEY,COUNT,MAXIT,STATUS
      REAL  A,B,X,XO,EPS
      DOUBLE  PRECISION  SUM
      INTRINSIC  ABS
      DIMENSION  A(10,10),B(10),X(10),XO(10)
* Initial values of X
      DO 10 I=1,N
        XO(I)  =  B(I)/A(I,I)
90    CONTINUE
      COUNT  =  1
99    KEY  =  0
* Computing values of X(I)
      DO 30  I  =  1,N
        SUM  =  B(I)
        DO 20  J  =  1,N
            IF(I.EQ.J)  GOTO  20
            SUM  =  SUM  -  A(I,J)  *  XO(J)
20      CONTINUE
        X(I)  =  SUM/A(I,I)
        IF(KEY  .EQ.  0)  THEN
* Testing for accuracy
            IF(ABS((X(I)-XO(I))/X(I))  .GT.  EPS)  THEN
              KEY  =  1
            ENDIF
        ENDIF
30    CONTINUE
      IF(KEY.EQ.1)  THEN
* Testing for convergence
      IF(COUNT  .EQ.  MAXIT)  THEN
        STATUS  =  2
        RETURN
      ELSE
        STATUS  =  1
        DO 40  I  =  1,N
          XO(I)  =  X(I)
40      CONTINUE
      ENDIF
        COUNT  =  COUNT+1
        GO  TO  11

      ENDIF

      RETURN
      END
* ------------ End of subroutine JACOBI -------------- *
```

***Test Run Results*** The program was used to solve the following system of equations:

$$3x_1 + x_2 = 5$$
$$x_1 - 3x_2 = 5$$

The interactive computer output is given below:

```
          SOLUTION BY JACOBI ITERATION
What is the size of the system(n)?
2
Input coefficients a(i,j), row-wise
one row on each line
3 1
1 -3
Input vector b
5 5
SOLUTION VECTOR X
     2.0000000 -9.999998E-001
ITERATIONS =  14

Stop - Program terminated.
```

Now, rearrange the equations as shown below and then use program JACIT to solve the system.

$$x_1 - 3x_2 = 5$$
$$3x_1 + x_2 = 5$$

The output now is as given below:

```
          SOLUTION BY JACOBI ITERATION
What is the size of the system(n)?
2
Input coefficients a(i, j), row-wise
one row on each line
1 -3
3 I
Input verctor b
5 5

NO CONVERGENCE IN      50 ITERATIONS

Stop - Program terminated.
```

Note that the same two equations, when their positions are interchanged, do not produce required results even after 50 iterations. Convergence is discussed in Section 8.5.

## 8.3 GAUSS-SEIDEL METHOD

Gauss-Seidel method is an improved version of Jacobi iteration method. In Jacobi method, we begin with the initial values

$$x_1^{(0)}, x_2^{(2)}, \dots, x_n^{(0)}$$

and obtain next approximation

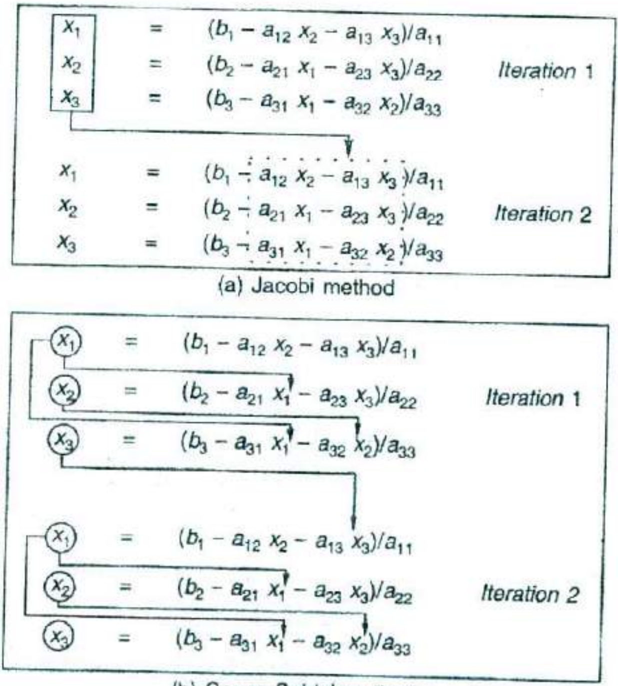
$$x_1^{(1)}, x_2^{(1)}, \dots, x_n^{(1)}$$

Note that, in computing $x_2^{(1)}$, we used $x_1^{(0)}$ and not $x_1^{(1)}$ which has just been computed. Since, at this point, both $x_1^{(0)}$ and $x_1^{(1)}$ are available, we can use $x_1^{(1)}$ which is a better approximation for computing $x_2^{(1)}$. Similarly, for computing $x_3^{(1)}$, we can use $x_1^{(1)}$ and $x_2^{(1)}$ along with $x_4^{(0)}, \dots, x_n^{(0)}$. This idea can be extended to all subsequent computations. This approach is called the *Gauss-Seidel* method.

The Gauss-Seidel method uses the most recent values of $x$ as soon as they become available at any point of iteration process. During the $(k+1)$th iteration of Gauss-Seidel method, $x_i$ takes the form

$$\boxed{x_i^{(k+1)} = \frac{b_i - \left(a_{i1}x_1^{(k+1)} + \dots + a_{ii-1}x_{i-1}^{(k+1)} + a_{ii+1}x_{i+1}^{(k)} \dots + a_{in}x_n^{(k)}\right)}{b_{ii}}} \tag{8.4}$$

When $i = 1$, all superscripts in the right-hand side become $(k)$ only. Similarly, when $i = n$, all become $(k + 1)$. Figure 8.1 illustrates pictorially the difference between the Jacobi and Gauss-Seidel method.


$x_1 = (b_1 - a_{12}x_2 - a_{13}x_3)/a_{11}$
$x_2 = (b_2 - a_{21}x_1 - a_{23}x_3)/a_{22}$
$x_3 = (b_3 - a_{31}x_1 - a_{32}x_2)/a_{33}$
Iteration 1
$x_1 = (b_1 - a_{12}x_2 - a_{13}x_3)/a_{11}$
$x_2 = (b_2 - a_{21}x_1 - a_{23}x_3)/a_{22}$
$x_3 = (b_3 - a_{31}x_1 - a_{32}x_2)/a_{33}$
Iteration 2

(a) Jacobi method

$x_1 = (b_1 - a_{12}x_2 - a_{13}x_3)/a_{11}$
$x_2 = (b_2 - a_{21}x_1 - a_{23}x_3)/a_{22}$
$x_3 = (b_3 - a_{31}x_1 - a_{32}x_2)/a_{33}$
Iteration 1
$x_1 = (b_1 - a_{12}x_2 - a_{13}x_3)/a_{11}$
$x_2 = (b_2 - a_{21}x_1 - a_{23}x_3)/a_{22}$
$x_3 = (b_3 - a_{31}x_1 - a_{32}x_2)/a_{33}$
Iteration 2

(b) Gauss-Seidel method

Fig. 8.1 Comparison of Jacobi and Gauss-Seidel methods

### Example 8.2

)btain the solution of the following system using Gauss-Seidel iteration method

$$\begin{aligned} 2x_1 + x_2 + x_3 &= 5 \\ 3x_1 + 5x_2 + 2x_3 &= 15 \\ 2x_1 + x_2 + 4x_3 &= 8 \end{aligned}$$

$$\begin{aligned} x_1 &= (5 - x_2 - x_3)/2 \\ x_2 &= (15 - 3x_1 - 2x_3)/5 \\ x_3 &= (8 - 2x_1 - x_2)/4 \end{aligned}$$

Assuming initial value as $x_1 = 0$, $x_2 = 0$, and $x_3 = 0$

*Iteration 1* $x_1 = (5 - 0 - 0)/2 = 2.5$

$x_2 = (15 - 3 \times 2.5 - 0)/5 = 1.5$

$x_3 = (8 - 2 \times 2.5 - 1.5)/4 = 0.4$ (rounded to one decimal)

*Iteration 2* $x_1 = (5 - 1.5 - 0.4)/2 = 1.6$

$x_2 = (15 - 3 \times 1.6 - 2 \times 0.4)/5 = 1.9$

$x_3 = (8 - 2 \times 1.6 - 1.9)/4 = 0.7$

We can continue this process until we get $x_1 = 1.0$, $x_2 = 2.0$ and $x_3 = 1.0$ (correct answers)

## Algorithm

Gauss-Seidel algorithm is a simple modification of the algorithm of the Jacobi method. Note that once a new value of$x_i^{(k+1)}$ has been calculated and compared with the previous values of $x_i^{(k)}$, the previous value is no longer required and, therefore, the previous value can be replaced by the new one. This implies that we need not use two vectors (one to store previous values and another to store new values) for storing $x$ values. We need to use only one vector $x$ that stores always the latest values of $x$. This is illustrated in Algorithm 8.2

**Gauss-Seidel method**

```
1. Obtain $n$, $a_{ij}$ and $b_i$ values
2. Set $x_i = b_i/a_{ii}$        for $i = 1$ to $n$
3. Set key = 0
4. For $i = 1$ to $n$
      (i) Set sum = $b_i$
     (ii) For $j = 1$ to n $(j \neq i)$
             Set sum = sum $- a_{ij} x_j$
          Repeat $j$
```

*(Contd.)*

*(Contd.)*

```
        (iii) Set dummy = sum / $a_{ii}$
        (iv) If key = 0 then
                 if $\left|\frac{\text{dummy} - x_i}{\text{dummy}}\right|$ > error then
                 set key = 1
        (v) Set $x_i$ = dummy
    Repeat i
5. If key = 1 then
        go to step 3
6. Write results
```

**Algorithm 8.2**

## Program GASIT

Like JACIT, the program GASIT also solves a system of $n$ linear equations but employs the Gauss-Seidel iteration method as detailed in Algorithm 8.2. The iteration algorithm is implemented with the help of a subprogram called GASEID.

```
* ---------------------------------------------------------- *
      PROGRAM GASIT
* ---------------------------------------------------------- *
* Main program                                                *
*   This program uses the subprogram GASEID to solve a        *
*   system of equations by Gauss-Seidel iteration method      *
* ---------------------------------------------------------- *
* Functions invoked                                           *
*     NIL                                                     *
* ---------------------------------------------------------- *
* Subroutines used                                            *
*     GASEID                                                  *
* ---------------------------------------------------------- *
* Variables used                                              *
*     A - Coefficient matrix                                  *
*     B - Right side vector                                   *
*     N - System size                                         *
*     X - Solution vector                                     *
*     COUNT - Number of iterations completed                  *
*     STATUS - Convergence status                             *
* ---------------------------------------------------------- *
* Constants used                                              *
*    EPS - Error bound                                        *
*    MAXIT - Maximum iterations permitted                     *
* ----------------------------------------------------------- *
      REAL A,B,X,EPS
      INTEGER N,COUNT,MAXIT,STATUS
      PARAMETER(EPS=0.000001,MAXIT=50)
      DIMENSION A(10,10), B(10), X(10)

      WRITE(*,*)
      WRITE(*,*)      'SOLUTION BY GAUSS-SEIDEL ITERATION'
      WRITE(*,*)

      WRITE(*,*) 'What is the size of the system(n)?'
      READ(*,*) N
      WRITE(*,*)    'Input coefficients a(i,j), row-wise'
      WRITE(*,*)    'one row on each line'

      DO 20 I = 1,N
         READ(*,*)  (A(I,J),J=1,N)
20    CONTINUE

      WRITE(*,*)     'Input vector b'
      READ(*,*)  (B(I), I = 1, N)

      CALL GASEID(N,A,B,X,EPS,COUNT,MAXIT,STATUS)

      IF(STATUS .EQ. 2) THEN
         WRITE(*,*)
         WRITE(*,*) 'NO CONVERGENCE IN', MAXIT,
                    'ITERATIONS'
         WRITE(*,*)
      ELSE
         WRITE(*,*)
         WRITE(*,*) 'SOLUTION VECTOR X'
         WRITE(*,*)
         WRITE(*,*) (X(I), I = 1, N)
         WRITE(*,*)
         WRITE(*,*) 'ITERATIONS = ', COUNT
         WRITE(*,*)
      ENDIF

      STOP
      END

* ------------End of main program GASIT ------------ *
* ------------------------------------------------- *
   SUBROUTINE GASEID(N,A,B,X,EPS,COUNT,MAXIT,STATUS)
* ------------------------------------------------- *
* Subroutine
*    This subroutine solves a system of linear
*    equations using Gauss-Seidel iteration algorithm
* ---------------------------------------------------
* Arguments
* Input
*    N - Number of equations
*    A - Coefficient matrix
*    B - Right side vector
*    EPS - Error bound
*    MAXIT - Maximum iterations allowed
* Output
*    X - Solution vector
*    COUNT - Number of iterations done
*    STATUS - Status of convergence
* -------------------------------------------------- *
* Local Variables                                    *
*    DUMMY,SUM,KEY                                   *
* -------------------------------------------------- *
* Functions invoked                                  *
*    ABS                                             *
* -------------------------------------------------- *
* Subroutines called                                 *
*    NIL                                             *
* -------------------------------------------------- *

      INTEGER N,KEY,COUNT,MAXIT,STATUS
      REAL A,B,X,EPS,DUMMY
      DOUBLE PRECISION SUM
      DIMENSION A(10,10),B(10),X(10),XO(10)
      INTRINSIC ABS

* Initial values of X
      DO 10 I = 1,N
       X(I) = B(I) / A(I,I)
10    CONTINUE

      COUNT = 1
11    KEY = 0

* Computing X(I) values
      DO 30 I = 1,N
          SUM = B(I)
          DO 20 J = 1,N
            IF(I.EQ.J) GOTO 20
            SUM = SUM - A(I,J) * X(J)
```

```
20      CONTINUE
        DUMMY = SUM/A(I,I)
        IF(KEY .EQ. 0) THEN
*         Testing for accuracy
          IF(ABS((DUMMY - X(I))/DUMMY) .GT. EPS) THEN
             KEY = 1
          ENDIF
          ENDIF
          X(I) = DUMMY
30    CONTINUE
      IF(KEY .EQ. 1) THEN
*        Testing for convergence
         IF(COUNT .EQ. MAXIT) THEN
           STATUS = 2
           RETURN
       ELSE
           STATUS = 1
           COUNT = COUNT + 1
           GOTO 11
       ENDIF
      ENDIF
      RETURN
      END
* ------------ End of subroutine GASEID--------------- *
```

***Test Run Results*** The program was used to solve two different sets of equations and the results are as follows:

```
First set
         SOLUTION BY GAUSS-SEIDEL ITERATION
 What is the size of the system(n)?
 3
 Input Coefficients a(i,j), row-wise
 one row on each line
 2 1 3
 4 4 7
 2 5 9
 Input vector b
 1 1 3
 SOLUTION VECTOR X
 -4.999992E-001   -9.999992E-001  9.999993E-001
 ITERATIONS = 38
 Stop - Program terminated.
```

```
Second set
          SOLUTION BY GAUSS-SEIDEL ITERATION
  What is the size of the system(n)?
  3
  Input coefficients a(i,j), row-wise
  one row on each line
  7 63 0
  3 30 0
  2 28 10
  Input vector b
  13.3 3.9 6.9

  NO CONVERGENCE IN          50 ITERATIONS

  Stop - Program terminated.
```

## 8.4 METHOD OF RELAXATION

Relaxation method represents a slightly modified version of the Gauss-Seidel method. The modification is aimed at faster convergence. The basic idea is to take the change produced in a Gauss-Seidel iteration step and extrapolate the new value by a factor $r$ of this change. The new relaxation value is given by

$$\boxed{\begin{aligned} x_{ir}^{(k+1)} &= x_i^{(k)} + r(x_i^{(k+1)} - x_i^{(k)}) \\ &= r x_i^{(k+1)} + (1-r)\, x_i^{(k)} \end{aligned}} \tag{8.5}$$

The parameter $r$ is called the *relaxation parameter*. This step is applied "successively" to each component of vector $\boldsymbol{x}$ during iteration process and, therefore, the method is known as *successive relaxation method*.

The parameter $r$ may be assigned a value between 0 and 2. We have the following possibilities:

| | |
|---|---|
| $0 < r < 1$ | under-relaxation |
| $r = 1$ | no relaxation ($x_{ir}^{(k+1)} = x_i^{(k+1)}$) |
| $1 < r < 2$ | over-relaxation |

For values of $r$ between 1 and 2, an extra weight is placed on the present value and Eq. (8.5) really represents an extrapolation. The intention here is to push the estimate closer to the solution. This method, when $1 < r < 2$, is popularly known as *successive over-relaxation* (or SOR) method. It is also known as *simultaneous* over-relaxation method.

The SOR technique can be easily implemented by a simple modification of the Gauss-Seidel algorithm. The relaxation value is obtained using Eq. (8.5) at the end of evaluation of each value of $\boldsymbol{x}$. The extrapolated value becomes the new value of $\boldsymbol{x}$ for the next cycle. Equation (8.5) can be simply implemented as

$$\boxed{x_i^{(k+1)} = r x_i^{(k+1)} + (1-r)\, x_i^{(k)}} \tag{8.6}$$

That is, the old value of $x_i^{(k+1)}$ is replaced by the new value of $x_i^{(k+1)}$. e implementation of this step is shown in Algorithm 8.3.

The choice of value of $r$ depends on the problem and is often decided empirically.

**SOR method**

```
Algorithm is the same as Algorithm 8.2, except the statement
    (iii) Set dummy = sum/$a_{ii}$
is replaced by a pair of statements
        Set dummy = sum/$a_{ii}$
        Set dummy = $r$ × dummy + $(1 - r)\, x_i$
```

**Algorithm 8.3**

## 8.5 CONVERGENCE OF ITERATION METHODS

### Condition for Convergence

We know that the iteration methods presented here are based on the basic idea of the fixed point method discussed in Chapter 6. We have shown that sufficient condition for convergence for solving one nonlinear equation is

$$|G'(x)| < 1$$

and for two nonlinear equations, $F(x, y)$ and $G(x, y)$, are

$$\left|\frac{\partial F}{\partial x}\right| + \left|\frac{\partial G}{\partial x}\right| < 1 \tag{8.7}$$

$$\left|\frac{\partial F}{\partial y}\right| + \left|\frac{\partial G}{\partial y}\right| < 1 \tag{8.8}$$

These conditions apply to linear equations as well. Therefore, we can use these conditions in the Jacobi and Gauss-Seidel iteration methods.

For the sake of simplicity, let us consider a two-equation linear system. We can express the Gauss-Seidel algorithm as follows:

$$x_1 = F(x_1, x_2) = \frac{1}{a_{11}}\,(b_1 - a_{12}x_2)$$

$$= \frac{b_1}{a_{11}} - \frac{a_{12}}{a_{11}}\,x_2 \tag{8.9}$$

$$x_2 = G(x_1, x_2) = \frac{1}{a_{22}}(b_2 - a_{21}x_1)$$

$$= \frac{b_2}{a_{22}} - \frac{a_{21}}{a_{22}} x_1 \tag{8.10}$$

The partial derivatives of these equations are

$$\frac{\partial F}{\partial x_1} = 0, \qquad \frac{\partial F}{\partial x_2} = -\frac{a_{12}}{a_{11}}$$

and

$$\frac{\partial G}{\partial x_1} = -\frac{a_{21}}{a_{22}}, \qquad \frac{\partial G}{\partial x_2} = 0$$

Substituting these values in Eqs (8.7) and (8.8), we get

$$\left|\frac{a_{21}}{a_{22}}\right| < 1 \qquad \text{and} \qquad \left|\frac{a_{12}}{a_{11}}\right| < 1$$

This means that

$$|a_{11}| > |a_{12}| \tag{8.11}$$

and

$$|a_{22}| > |a_{21}| \tag{8.12}$$

That is, the absolute value of diagonal element must be greater than that of the off-diagonal element for each row.

The above derivation can be extended to a general system of $n$ equations to show that

$$\boxed{|a_{ii}| > \sum_{j=1}^{n} |a_{ij}|, \qquad i \neq j} \tag{8.13}$$

For each row, the absolute value of the diagonal element should be greater than the sum of absolute values of the other elements in the equation. Remember that this condition is sufficient, but not necessary, for convergence. Some systems may converge even if this condition is not satisfied.

Systems that satisfy the condition Eq. (8.13) are called *diagonally dominant* systems. Convergence of such systems are guaranteed.

## Rate of Convergence

Consider the iterative Eqs (8.9) and (8.10). At ($k$+1)th iteration, we have

$$x_1^{(k+1)} = \frac{b_1}{a_{11}} - \frac{a_{12}}{a_{11}} x_2^k \tag{8.14}$$

$$x_2^{(k+1)} = \frac{b_2}{a_{22}} - \frac{a_{21}}{a_{22}} x_1^{k+1} \tag{8.15}$$

Substituting for $x_1^{(k+1)}$ in Eq. (8.15), we get

$$x_2^{(k+1)} = \frac{b_2}{a_{22}} - \frac{a_{21}}{a_{22}}\left[\frac{b_1}{a_{11}} - \frac{a_{12}}{a_{11}} x_2^k\right] \tag{8.16}$$

Similarly, we have

$$x_2^{(k+2)} = \frac{b_2}{a_{22}} - \frac{a_{21}}{a_{22}}\left[\frac{b_1}{a_{11}} - \frac{a_{12}}{a_{11}} x_2^{k+1}\right] \tag{8.17}$$

Subtracting Eq. (8.16) from Eq. (8.17), we get

$$x_2^{(k+2)} - x_2^{(k+1)} = \frac{a_{12}a_{21}}{a_{11}a_{22}}\,(x_2^{(k+1)} - x_2^{(k)})$$

If we denote the errors as

$$e_2^{k-1} = x_2^{(k+2)} - x_2^{(k+1)}$$

$$e_2^k = x_2^{(k+1)} - x_2^{(k)}$$

Then

$$\boxed{e_2^{(k+1)} = \frac{a_{12}a_{21}}{a_{11}a_{22}}\,e_2^{(k)}} \tag{8.18}$$

If we want the error to decrease with successive iterations, then we should have the coefficients such that

$$\frac{a_{12}a_{21}}{a_{11}a_{22}} < 1 \tag{8.19}$$

This also conforms with Eqs (8.11) and (8.12).

### Example 8.3

Solve the equations

$$3x_1 + x_2 = 5$$

$$x_1 - 3x_2 = 5$$

by the Gauss-Seidel method

First, we rearrange the equations in the form

$$x_1 = 1/3(5 - x_2)$$

$$x_2 = 1/3(x_1 - 5)$$

Assuming initial values as $x_1^0 = 0$, and $x_2^0 = 0$

$$x_1^{(1)} = 5/3$$

Remember, the new value of $x$, should be used in the calculation of new $x_2$. Therefore

$$x_2^{(1)} = -10/9$$

Similarly,

$$x_1^{(2)} = \frac{1}{3}\left(5 + \frac{10}{9}\right) = \frac{55}{27}$$

The table below shows the values of $x_1$ and $x_2$ rounded to 4 decimal places.

| *Iteration* | $x_1$ | $x_2$ | *True error in* $x_1$ | *True error in* $x_2$ |
|---|---|---|---|---|
| 0 | 0.0000 | 0.0000 | 2.0000 | 1.0000 |
| 1 | 1.6667 | -1.1111 | 0.3333 | 0.1111 |
| 2 | 2.0370 | -0.9877 | 0.0370 | 0.0123 |
| 3 | 1.9959 | -1.0014 | 0.0041 | 0.0014 |
| 4 | 2.0005 | -0.9999 | 0.0005 | 0.0001 |
| 5 | 2.0000 | -1.0000 | 0.0000 | 0.0000 |

The process converges to the solution ($x_1 = 2$, $x_2 = -1$) in five iterations. Note that the given system is *diagonally dominant*. The convergence is graphically illustrated in Fig. 8.2

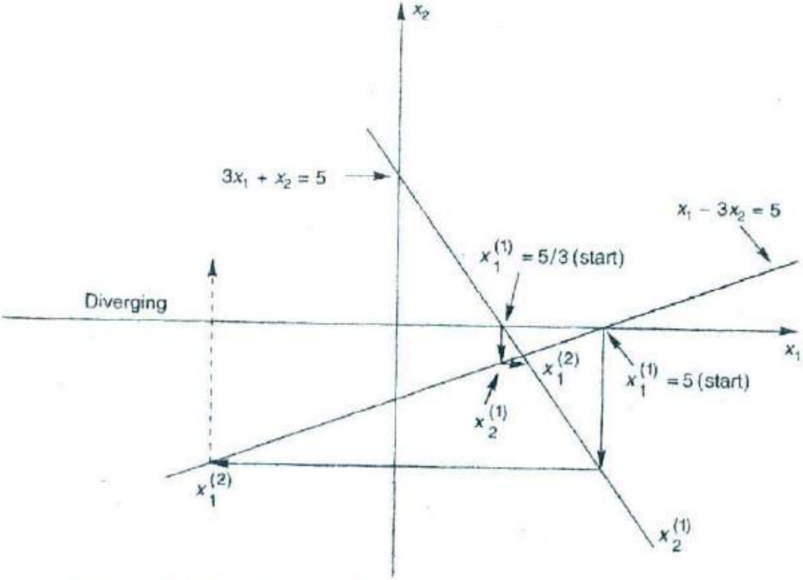


**Fig. 8.2** Pictorial representation of Gauss-Seidel convergence

### Example 8.4

Solve the equations

$$x_1 - 3x_2 = 5$$
$$3x_1 + x_2 = 5$$

by the Gauss-Seidel method

Note that the system contains the same two equations as in Example 8.3, except they are interchanged.

The iterative equations are

$$x_1 = 5 + 3x_2$$
$$x_2 = 5 - 3x_1$$

As before, we start with $x_1^0 = 0$ and $x_2^0 = 0$. Then,

$$x_1^{(1)} = 5 \quad \text{and} \quad x_2^{(1)} = -10$$
$$x_1^{(2)} = -25 \quad \text{and} \quad x_2^{(2)} = 80$$
$$x_1^{(3)} = 245 \quad \text{and} \quad x_2^{(3)} = -730$$

It is clear that the process does not converge towards the solution. Rather, it diverges (see Fig. 8.2). The result will be the same even if we start with the initial values very close to the solution (except the solution itself). Readers may try with $x_1^0 = 2.5$ and $x_2^0 = -1.2$.

From Examples 8.3 and 8.4 we observe the following:

1. Iteration process converges when

$$\left|\frac{a_{21}}{a_{22}}\right| < 1 \quad \text{and} \quad \left|\frac{a_{12}}{a_{11}}\right| < 1$$

2. The process does not converge for the same set of equations when their order is changed. That is, when

$$\frac{a_{12}a_{21}}{a_{11}a_{22}} > 1$$

the process does not converge

3. When it converges, the errors in $x_1$ and $x_2$ decrease by a factor of

$$\frac{a_{11}a_{22}}{a_{12}a_{21}} = 9$$

at each iteration

4. Stronger the diagonal elements, faster the convergence.

## 8.6 SUMMARY

Iterative methods provide an alternative to the direct methods for solving linear equations. These methods are particularly suitable for solving ill-conditioned systems. We considered the following three iterative methods:

- Jacobi method
- Gauss-Seidel method
- Successive Over Relaxation (SOR) method

We also presented FORTRAN programs along with test results for the Jacobi and Gauss-Seidel methods.

We have shown that a sufficient condition for convergence is that, for each row, the absolute value of the diagonal element should be greater than the sum of absolute values of the other elements in the equation.

### Key Terms

*Diagonally dominant system*
*Gauss-Seidel iteration*
*Jacobi iteration*
*Relaxation parameter*
*Successive over relaxation*
*Successive relaxation method*

### REVIEW QUESTIONS

1. State the two popular approaches available for solving a system of linear equations.
2. What are the limitations and pitfalls of using direct methods for solving a system of linear equations?
3. State the two important factors that are to be considered while applying iterative methods.
4. The basic idea behind the Jacobi iterative method is essentially the same as that of fixed point method used for solving nonlinear equations. Explain.
5. Gauss-Seidel method is similar in principle to Jacobi method. Then, what is the difference between them?
6. Show that, for a two-equation system

$$a_{11}x_1 + a_{12}x_2 = b_1$$
$$a_{21}x_1 + a_{22}x_2 = b_2$$

   a sufficient condition for convergence of the iteration process is

$$\left|\frac{a_{12}a_{21}}{a_{11}a_{22}}\right| < 1$$

7. Explain the basic concept used in the relaxation method.
8. What is relaxation parameter?

9. What is meant by over-relaxation and under-relaxation?
10. Give an algorithm for solving a system of linear equations using the successive over-relaxation (SOR) method.

## REVIEW EXERCISES

1. Solve the set of equations given below by Jacobi method.

NUB

$$3x_1 - 6x_2 + 2z = 15$$
$$4x_1 - x_2 + z = 2$$
$$x_1 - 3x_2 + 7z = 22$$

2. Solve the system of equations

NUB

$$2x - y + 2z = 6$$
$$2x - y + z = 3$$
$$x + 3y - z = 4$$

by using Jacobi method.

3. Solve the systems given in Exercises 1 and 2 by Gauss-Seidel iteration. Compare the rate of convergence in both the cases.
4. Solve the pair equations

$$x_1 + 2x_2 = 5$$
$$3x_1 + x_2 = 5$$

by applying Jacobi method to the equations

$$x_1 = 5 - 2x_2$$
$$x_2 = 5 - 3x_1$$

Observe the divergence.

5. Solve the equations in Exercise 4 applying Gauss-Seidel method. Compare the divergence with that of earlier one.
6. Interchange the order of equations given in Exercise 4 and then solve them
   (a) using Jacobi method
   (b) using Gauss-Seidel method

   Compare the convergence.
7. Solve the system of equations

$$3x_1 - 2x_2 = 5$$
$$-x_1 + 2x_2 - x_3 = 0$$
$$-2x_2 + x_3 = -1$$

by applying
   (a) Jacobi method
   (b) Gauss-Seidel method, and
   (c) Successive over-relaxation method with $r = 1.4$

Comment on the results.

8. Solve the following equations by Gauss-Seidel method

$$
\begin{aligned}
2x - 7y - 10z &= -17\\
5x + y + 3z &= 14\\
x + 10y + 5z &= 7
\end{aligned}
$$

Assume suitable initial values.

9. Monthly faculty salary in three departments of an institute is given below. Assuming that the salary for a particular category is same in all the departments, calculate the salary of each category of faculty.

| *Department* | *Number of Faculty* | | | *Total Salary* |
|---|---|---|---|---|
| | *Professor* | *Asst. Professor* | *Lecturer* | *(in '000)* |
| A | 2 | 2 | 4 | 60 |
| B | 3 | 1 | 2 | 50 |
| C | 1 | 4 | 3 | 60 |

10. Mr. Ram has invested a sum of Rs 20,000 in three types of fixed deposits with an interest rate of 10%, 11% and 12%. He earns an annual interest of Rs 2,220 from all the three types of deposits. If sum of the amounts with 11% and 12% interest rates is four times the amount earning 10% interest, what is the amount invested in each type.

## PROGRAMMING PROJECTS

1. Develop a menu-driven, user-friendly single program which provides options for using either Jacobi method or Gauss-Seidel method.
2. Modify the Gauss-Seidel iteration program to incorporate the successive over relaxation method to improve the speed of convergence.

CHAPTER 9

# Curve Fitting: Interpolation

## 9.1 INTRODUCTION

Scientists and engineers are often faced with the task of estimating the value of dependent variable $y$ for an intermediate value of the independent variable $x$, given a table of discrete data points $(x_i, y_i)$, $i = 0,1,...n$. This task can be accomplished by constructing a function $y(x)$ that will pass through the given set of points and then evaluating $y(x)$ for the specified value of $x$. The process of construction of $y(x)$ to fit a table of data points is called *curve fitting*. A table of data may belong to one of the following two categories:

1. *Table of values of well-defined functions:* Examples of such tables are logarithmic tables, trigonometric tables, interest tables, steam tables, etc.
2. *Data tabulated from measurements made during an experiment:* In such experiments, values of the dependent variable are recorded at various values of the independent variable. There are numerous examples of such experiments—the relationship between stress and strain on a metal strip, relationship between voltage applied and speed of a fan, relationship between time and temperature raise in heating a given volume of water, relationship between drag force and velocity of a falling body, etc., can be tabulated by suitable experiments.

In category 1, the table values are accurate because they are obtained from well-behaved functions. This is not the case in category 2 where the relationship between the variables is not well-defined. Accordingly, we have two approaches for fitting a curve to a given set of data points.

In the first case, the function is constructed such that it passes through all the data points. This method of constructing a function and estimating values at non-tabular points is called *interpolation*. The functions are known as *interpolation polynomials*.

In the second case, the values are not accurate and, therefore, it will be meaningless to try to pass the curve through every point. The best strategy would be to construct a single curve that would represent the general trend of the data, without necessarily passing through the individual points. Such functions are called *approximating functions*. One popular approach for finding an approximate function to fit a given set of experimental data is called *least-squares regression*. The approximating functions are known as *least-squares polynomials*.

Figure 9.1 shows an approximate linear function and an interpolation polynomial for a set of data. Note that although the interpolation polynomial passes through all the points, the curve oscillates widely at the end and beyond the range of data. The linear approximating curve which does not pass through any of the points appears to represent the trend of data adequately. The straight line gives a much better idea of likely values beyond the table points.

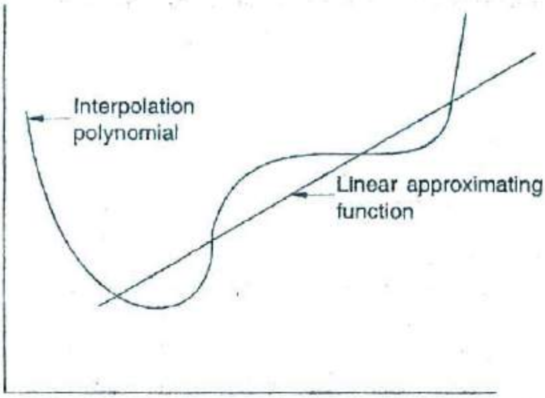


**Fig. 9.1** Curve fitting to a set of points

In this chapter, we discuss various methods of interpolation. They include:

1. Lagrange interpolation
2. Newton's interpolation
3. Newton-Gregory forward interpolation
4. Spline interpolation

Before we discuss these methods, we introduce various forms of polynomials that are used in deriving interpolation functions. Least-squares regression techniques are discussed in the next chapter.

## 9.2 POLYNOMIAL FORMS

The most common form of an $n$th order polynomial is

$$\boxed{p(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n} \tag{9.1}$$

This form, known as the *power form*, is very convenient for differentiating and integrating the polynomial function and, therefore, are most widely used in mathematical analysis. However, there are situations where this form has been found inadequate, as illustrated by Example 9.1.

### Example 9.1

Consider the power form of $p(x)$ for $n = 1$,

$$p(x) = a_0 + a_1 x.$$

Given that

$$p(100) = +3/7$$
$$p(101) = -4/7$$

obtain the linear polynomial $p(x)$ using four-digit floating point arithmetic. Verify the polynomial by substituting back the values $x = 100$ and $x = 101$.

$$p(100) = a_0 + 100\, a_1 = +0.4286$$
$$p(101) = a_0 + 101\, a_1 = -0.5714$$

Then, we get

$$a_1 = -1$$
$$a_0 = 100.4 \text{ (only four significant digits)}$$

Therefore,

$$p(x) = 100.4 - x$$

using this polynomial, we obtain

$$p(100) = 0.4$$
$$p(101) = -0.6$$

Compare these results with the original values of $p(100)$ and $p(101)$. We have lost three decimal digits.

Example 9.1 shows that the polynomials obtained using the power form may not always produce accurate results. In order to overcome such problems, we have alternative forms of representing a polynomial. One of them is the *shifted power form* as shown below:

$$\boxed{p(x) = a_0 + a_1 (x - C) + a_2 (x - C)^2 + \ldots + a_n (x - C)^n} \tag{9.2}$$

where $C$ is a point somewhere in the interval of interest. This form of representation significantly improves the accuracy of the polynomial evaluation. This is illustrated by Example 9.2.

### Example 9.2

Repeat Example 9.1 using the shifted power form and four-digit arithmetic.

Shifted power form of first order $p(x)$ is

$$p(x) = a_0 + a_1 (x - C)$$

Let us choose the centre $C$ as 100. Then

$$p(x) = a_0 + a_1 (x - 100)$$

This gives,

$$p(100) = a_0 = 3/7 = 0.4286$$

$$p(101) = 0.4286 + a_1 (101 - 100) = -0.5714$$

$$a_1 = -1$$

Thus the linear polynomial becomes

$$p(x) = 0.4286 - (x - 100)$$

Using this polynomial, we obtain

$$p(100) = 0.4286$$

$$p(101) = -0.5714$$

Note the improvement in the results.

Note that Eq. (9.2) is the *Taylor expansion* of $p(x)$ around the point $C$, when the coefficients $a_i$ are replaced by appropriate function derivatives. It can be easily verified that

$$a_i = \frac{p^{(i)}(C)}{i!} \quad i = 0, 1, 2, \dots n$$

where $p^{(i)}(C)$ is the $i$th derivative of $p(x)$ at $C$.

There is a third form of $p(x)$ known as *Newton form*. This is a generalised shifted power form as shown below:

$$\boxed{\begin{aligned} p(x) = a_0 + a_1 (x - C_1) + a_2 (x - C_1)(x - C_2) + a_3 (x - C_1) \\ (x - C_2)(x - C_3) + \dots + a_n (x - C_1)(x - C_2) \dots (x - C_n) \end{aligned}} \tag{9.3}$$

Note that Eq. (9.3) reduces to shifted power form when $C_1 = C_2 = C_3 = \dots = C_n$ and to simple power form when $C_i = 0$ for all $i$. The Newton form plays an important role in the derivation of an interpolating polynomial as seen in Section 9.5.

Polynomials can also be expressed in the form

$$\begin{aligned} p_2(x) = b_0 &(x - x_1)(x - x_2) \\ &+ b_1 (x - x_0)(x - x_2) \\ &+ b_2 (x - x_0)(x - x_1) \end{aligned}$$

In general form,

$$\boxed{P_n(x) = \sum_{i=0}^{n} b_i \prod_{j=0,\, j \neq i}^{n} (x - x_j)} \tag{9.4}$$

## 9.3 LINEAR INTERPOLATION

The simplest form of interpolation is to approximate two data points by a straight line. Suppose we are given two points $(x_1, f(x_1))$ and $(x_2, f(x_2))$. These two points can be connected linearly as shown in Fig. 9.2. Using the concept of similar triangles, we can show that

$$\frac{f(x) - f(x_1)}{x - x_1} = \frac{f(x_2) - f(x_1)}{x_2 - x_1}$$

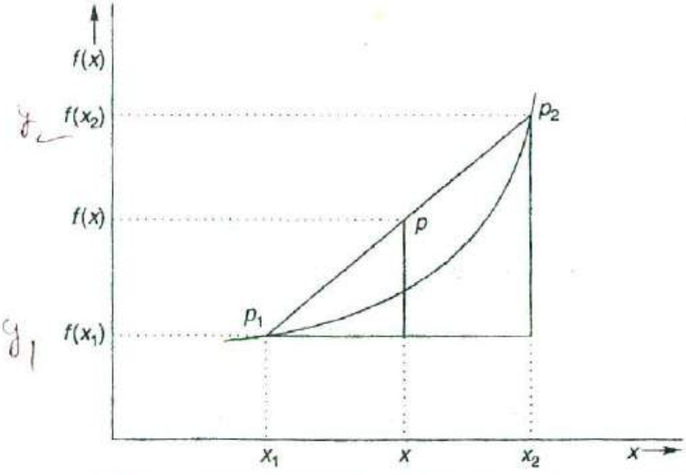


**Fig. 9.2** Graphical representation of linear interpolation

Solving for $f(x)$, we get

$$\boxed{f(x) = f(x_1) + (x - x_1)\,\frac{f(x_2) - f(x_1)}{x_2 - x_1}} \tag{9.5}$$

Equation (9.5) is known as *linear interpolation formula*. Note that the term

$$\frac{f(x_2) - f(x_1)}{x_2 - x_1}$$

represents the slope of the line. Further, note the similarity of equation (9.5) with the *Newton form* of polynomial of first-order.

$$C_1 = x_1$$

$$a_0 = f(x_1)$$

$$a_1 = \frac{f(x_2) - f(x_1)}{x_2 - x_1}$$

The coefficient $a_1$ represents the first derivative of the function.

**Example 9.3**

The table below gives square roots for integers.

| $x$ | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| $f(x)$ | 1 | 1.4142 | 1.7321 | 2 | 2.2361 |

Determine the square root of 2.5

The given value of 2.5 lies between the points 2 and 3. Therefore,

$$x_1 = 2, \qquad f(x_1) = 1.4142$$

$$x_2 = 3, \qquad f(x_2) = 1.7321$$

Then

$$f(2.5) = 1.4142 + (2.5 - 2.0)\ \frac{1.7321 - 1.4142}{3.0 - 2.0}$$

$$= 1.4142 + (0.5)\ (0.3179)$$

$$= 1.5732$$

The correct answer is 1.5811. The difference is due to the use of a linear model to a nonlinear one.

Now, let us repeat the procedure assuming $x_1 = 2$ and $x_2 = 4$.

$$f(x_1) = 1.4142$$

$$f(x_2) = 2.0$$

Then,

$$f(2.5) = 1.4142 + (2.5 - 2.0)\ \frac{2.0 - 1.4142}{4.0 - 2.0}$$

$$= 1.4142 + (0.5)\,(0.2929)$$
$$= 1.5607$$

Notice that the error has increased from 0.0079 to 0.0204. In general, the smaller the interval between the interpolating data points, the better will be the approximation.

The results could be improved considerably by using higher-order interpolation polynomials. We shall demonstrate this in the next section.

## 9.4 LAGRANGE INTERPOLATION POLYNOMIAL

In this section, we derive a formula for the polynomial of degree $n$ which takes specified values at a given set of $n + 1$ points.

Let $x_0, x_1, \ldots x_n$ denote $n$ distinct real numbers and let $f_0, f_1, \ldots, f_n$ be arbitrary real numbers. The points $(x_0, f_0), (x_1, f_1), \ldots (x_n, f_n)$ can be imagined to be data values connected by a curve. Any function $p(x)$ satisfying the conditions

$$p(x_k) = f_k \qquad \text{for} \qquad k = 0, 1, \ldots n$$

is called an *interpolation function*. An interpolation function is, therefore, a curve that passes through the data points as pointed out in Section 9.1.

Let us consider a second-order polynomial of the form

$$\begin{aligned} p_2(x) = {} & b_1(x - x_0)(x - x_1) \\ & + b_2(x - x_1)(x - x_2) \\ & + b_3(x - x_2)(x - x_0) \end{aligned} \tag{9.6}$$

If $(x_0, f_0)$, $(x_1, f_1)$ and $(x_2, f_2)$ are the three interpolating points, then we have

$$\begin{aligned} p_2(x_0) &= f_0 = b_2(x_0 - x_1)(x_0 - x_2) \\ p_2(x_1) &= f_1 = b_3(x_1 - x_2)(x_1 - x_0) \\ p_2(x_2) &= f_2 = b_1(x_2 - x_0)(x_2 - x_1) \end{aligned}$$

Substituting for $b_1$, $b_2$ and $b_3$ in Eq. (9.6), we get

$$\begin{aligned} p_2(x) = {} & f_0 \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} \\ & + f_1 \frac{(x - x_2)(x - x_0)}{(x_1 - x_2)(x_1 - x_0)} \\ & + f_2 \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)} \end{aligned} \tag{9.7}$$

Equation (9.7) may be represented as

$$p_2(x) = f_0\, l_0(x) + f_1\, l_1(x) + f_2\, l_2(x)$$

$$= \sum_{i=0}^{2} f_i l_i(x)$$

where

$$l_i(x) = \prod_{j=0,\, j \neq i}^{2} \frac{(x - x_j)}{(x_i - x_j)}$$

In general, for $n+1$ points we have $n$th degree polynomial as

$$\boxed{p_n(x) = \sum_{i=0}^{n} f_i l_i(x)} \tag{9.8}$$

where

$$\boxed{l_i(x) = \prod_{j=0,\, j \neq i}^{n} \frac{(x - x_j)}{(x_i - x_j)}} \tag{9.9}$$

Equation (9.8) is called the *Lagrange interpolation polynomial*. The polynomials $l_i\ (x)$ are known as *Lagrange basis polynomials*. Observe that

$$l_i(x_j) = \begin{cases} 1 & \text{for } i = j \\ 0 & \text{for } i \neq j \end{cases}$$

Now, consider the case $n = 1$

$$l_0(x) = \frac{x - x_1}{x_0 - x_1}$$

$$l_1(x) = \frac{x - x_0}{x_1 - x_0}$$

Therefore,

$$p_1(x) = f_0 \frac{x - x_1}{x_0 - x_1} + f_1 \frac{x - x_0}{x_1 - x_0}$$

$$= \frac{f_0(x - x_1) - f_1(x - x_0)}{x_0 - x_1}$$

$$= f_0 + \frac{f_1 - f_0}{x_1 - x_0}(x - x_0)$$

This is the *linear interpolation formula.*

**Example 9.4**

Consider the problem in Example 9.3. Find the square root of 2.5 using the second order Lagrange interpolation polynomial.

Let us consider the following three points:

$$x_0 = 2, \qquad x_1 = 3, \qquad \text{and} \qquad x_2 = 4$$

Then

$$f_0 = 1.4142, \qquad f_1 = 1.7321, \qquad \text{and} \qquad f_2 = 2$$

For x = 2.5, we have

$$l_0(2.5) = \frac{(2.5-3.0)(2.5-4.0)}{(2.0-3.0)(2.0-4.0)} = 0.3750$$

$$l_1(2.5) = \frac{(2.5-2.0)(2.5-4.0)}{(3.0-4.0)(3.0-2.0)} = 0.7500$$

$$l_2(2.5) = \frac{(2.5-2.0)(2.5-3.0)}{(4.0-2.0)(4.0-3.0)} = -0.125$$

$$p_2(2.5) = (1.4142)(0.3750) + (1.7321)(0.7500) + (2.0)(-0.125)$$
$$= 0.5303 + 1.2991 - 0.250 = 1.5794$$

The error is 0.0017 which is much less than the error obtained in Example 9.3

### Example 9.5

F

Find the Lagrange interpolation polynomial to fit the following data.

| $i$ | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| $x_i$ | 0 | 1 | 2 | 3 |
| $e^{x_i} - 1$ | 0 | 1.7183 | 6.3891 | 19.0855 |

→ f( )

Use the polynomial to estimate the value of $e^{1.5}$.

Lagrange basis polynomials are

$$l_0(x) = \frac{(x-1)(x-2)(x-3)}{(0-1)(0-1)(0-3)}$$

$$= \frac{x^3 - 6x^2 + 11x - 6}{-3}$$

$$l_1(x) = \frac{(x-0)(x-2)(x-3)}{(1-0)(1-2)(1-3)}$$

$$= \frac{x^3 - 5x^2 + 6x}{2}$$

$x_0 = 0$
$x_1 = 1$
$x_2 = 2$
$x_3 = 3$

$f_0 = 0$
$f_1 = 1.7183$
$f_2 = 6.3891$
$f_3 = 19.0855$

$x(x-1)(x-3)$
$= x(x^2-3x-x+3)$
$= x^3-3x^2-x^2+3x$
$= x^3-4x^2+3x$

$$l_2(x) = \frac{(x-0)\,(x-1)\,(x-3)}{(2-0)\,(2-1)\,(2-3)}$$

$$= \frac{x^3 - 4x^2 + 3x}{-2}$$

$$l_3(x) = \frac{(x-0)\,(x-2)\,(x-3)}{(3-0)\,(3-1)\,(3-2)}$$

$$= \frac{x^3 - 3x^2 + 2x}{6}$$

$$\boxed{p(1,5) = 0.9375}$$

The interpolation polynomial is

$$p(x) = f_0\, l_0(x) + f_1\, l_1(x) + f_2\, l_2(x) + f_3\, l_3(x)$$

$$= 0 + \frac{1.7183\,(x^3 - 5x^2 + 6x)}{2}$$

$$+ \frac{6.3891\,(x^3 - 4x^2 + 3x)}{-2}$$

$$+ \frac{19.0856\,(x^3 - 3x^2 + 2x)}{6}$$

$$= \frac{5.0732\,x^3 - 6.3621\,x^2 + 11.5987x}{6}$$

$$= 0.8455x^3 - 1.0604\,x^2 + 1.9331x$$

$$p(1.5) = 3.3677$$

$$e^{1.5} = p(1.5) + 1 = 4.3677$$

Points to be noted about Lagrange polynomial:

1. It requires $2(n+1)$ multiplications/divisions and $2n+1$ additions and subtractions
2. If we want to add one more data point, we have to compute the polynomial from the beginning. It does not use the polynomial already computed. That is, $p_{k+1}(x)$ does not use $p_k(x)$ which is already available

## Program LAGRAN

Program LAGRAN computes the interpolation value at a specified point, given a set of data points, using the Lagrange interpolation polynomial representation.

```
* ------------------------------------------------------------ *
     PROGRAM LAGRAN
* ------------------------------------------------------------ *
* Main program                                                 *
*    This program computes the interpolation value at a        *
*    specified point, given a set of data points, using        *
*    the Lagrange interpolation representation                 *
* ------------------------------------------------------------ *
* Functions invoked                                            *
*     NIL                                                      *
* ------------------------------------------------------------ *
* Subroutines used                                             *
*     NIL                                                      *
* ------------------------------------------------------------ *
* Variables used                                               *
*      XN  - Number of data sets                               *
*      X(I)- Data points                                       *
*      F(I)- Function values at data points                    *
*      XP  - Point at which interpolation is required          *
*      FP  - Interpolated value at XP                          *
*      LF  - Lagrangian factor.                                *
* ------------------------------------------------------------ *
* Constants used                                               *
*     MAX - Maximum number of data points permitted            *
* ------------------------------------------------------------ *

      INTEGER N,MAX
      REAL X,F,FP,LF,SUM
      PARAMETER(MAX = 10)
      DIMENSION X(MAX),F(MAX)
      WRITE(*,*)     'Input number of data points(N)'
      READ(*,*) N
      WRITE(*,*)     'Input data points X(I) and Function',
     +        'values F(I)'
      WRITE(*,*) 'one set in each line'
      DO 10 I = 1,N
         READ(*,*) X(I), F(I)
10    CONTINUE
      WRITE(*,*)     'Input X value at which'
      WRITE (*,*)    'interpolation is required'
      READ(*,*) XP
      SUM = 0.0
      DO 30 I = 1,N
        LF = 1.0
        DO 20 J = 1,N
```

```
            IF (I.NE.J) THEN
              LF = LF * (XP - X(J)) / (X(I) - X(J))
            ENDIF
20        CONTINUE
          SUM = SUM + LF * F(I)
30      CONTINUE
        FP = SUM

        WRITE(*,*)
        WRITE(*,*) 'LAGRANGIAN INTERPOLATION'
        WRITE(*,*)
        WRITE(*,*) 'Interpolated Function Value'
        WRITE(*,*) 'at X = ', XP, ' is', FP
        WRITE(*,*)

        STOP
        END
* --------------- End of main LAGRAN ----------------*
```

***Test Run Results*** The program was used to compute the function value at $x = 2.5$ for the following table of data points:

| $x$ | 2 | 3 | 4 |
|---|---|---|---|
| $f$ | 1.4142 | 1.7321 | 2.0 |

The results are shown below:

```
Input number of data points(N)
3
Input data points X(I) and Function values F(I)
one set in each line
2 1.4142
3 1.7321
4 2.0
Input X value at which
interpolation is required
2.5

LAGRANGIAN INTERPOLATION

Interpolated Function Value
at X = 2.5000000 is 1.5794000

Stop - Program terminated.
```

## NEWTON INTERPOLATION POLYNOMIAL

We have seen that, in Lagrange interpolation, we cannot use the work that has already been done if we want to incorporate another data point in order to improve the accuracy of estimation. It is therefore necessary to look for some other form of representation to overcome this drawback.

Let us now consider another form of polynomial known as *Newton form* which was discussed in Section 9.2. The Newton form of polynomial is

$$p_n(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + \dots + a_n(x - x_0)(x - x_1) \dots (x - x_{n-1}) \tag{9.10}$$

where the interpolation points $x_0, x_1, \dots x_{n-1}$ act as centres.

To construct the interpolation polynomial, we need to determine the coefficients $a_0, a_1, \dots a_n$. Let us assume that $(x_0, f_0), (x_1, f_1), \dots (x_{n-1}, f_{n-1})$ are the interpolating points. That is,

$$p_n(x_k) = f_k \qquad k = 0, 1, \dots n - 1$$

Now, at $x = x_0$, we have (using Eq. (9.10))

$$p_n(x_0) = \boxed{a_0 \;=\; f_0} \tag{9.11}$$

Similarly, at $x = x_1$,

$$p_n(x_1) = a_0 + a_1(x_1 - x_0) = f_1$$

Substituting for $a_0$ from Eq. (9.11), we get

$$\boxed{a_1 = \frac{f_1 - f_0}{x_1 - x_0}} \tag{9.12}$$

At $x = x_2$,

$$p_n(x_2) = a_0 + a_1(x_2 - x_0) + a_2(x_2 - x_0)(x_2 - x_1) = f_2$$

Substituting for $a_0$ and $a_1$ from Eqs. (9.11) and (9.12) and rearranging the terms, we get

$$\boxed{a_2 = \frac{[(f_2 - f_1)/(x_2 - x_1)] - [(f_1 - f_0)/(x_1 - x_0)]}{x_2 - x_0}} \tag{9.13}$$

Let us define a notation

$$f[x_k] = f_k$$

$$f[x_k, x_{k+1}] = \frac{f[x_{k+1}] - f[x_k]}{x_{k+1} - x_k}$$

$$f[x_k, x_{k+1}, x_{k+2}] = \frac{f[x_{k+1}, x_{k+2}] - f[x_k, x_{k+1}]}{x_{k+2} - x_k}$$

$$f[x_k, x_{k+1}, \dots x_i, x_{i+1}] = \frac{f[x_{k+1} \dots x_{i+1}] - f[x_k \dots x_i]}{x_{i+1} - x_k} \tag{9.14}$$

These quantities are called *divided differences*. Now we can express the coefficients $a_i$ in terms of these divided differences.

$$a_0 = f_0 = f[x_0]$$

$$a_1 = \frac{f_1 - f_0}{x_1 - x_0} = f[x_0, x_1]$$

$$a_2 = \frac{\dfrac{f_2 - f_1}{x_2 - x_1} - \dfrac{f_1 - f_0}{x_1 - x_0}}{x_2 - x_0}$$

$$= \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0}$$

$$= f[x_0, x_1, x_2]$$

Thus,

$$a_n = f[x_0, x_1, \ldots x_n] \tag{9.15}$$

Note that $a_1$ represents the *first divided difference* and $a_2$ the *second divided difference* and so on.

Substituting for $a_i$ coefficients in equation (9.10), we get

$$p_n(x) = f[x_0] + f[x_0, x_1](x - x_0) + f[x_0, x_1, x_2](x - x_0)(x - x_1) + \ldots + f[x_0, x_1, \ldots x_n](x - x_0)(x - x_1) \ldots (x - x_{n-1})$$

This can be written more compactly as

$$\boxed{p_n(x) = \sum_{i=1}^{n} f[x_0, \ldots x_i] \prod_{j=0}^{i-1} (x - x_j)} \tag{9.16}$$

Equation (9.16) is called *Newton's divided difference interpolation polynomial*.

### Example 9.6

Given below is a table of data for log $x$. Estimate log 2.5 using second order Newton interpolation polynomial.

| $i$ | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| $x_i$ | 1 | 2 | 3 | 4 |
| log $x_i$ | 0 | 0.3010 | 0.4771 | 0.6021 |

Second order polynomials require only three data points. We use the first three points

$$a_0 = f[x_0] = 0$$

$$a_1 = f[x_0, x_1] = \frac{f(x_1) - f(x_0)}{x_1 - x_0} = \frac{0.3010}{2-1} = 0.3010$$

$$a_2 = f[x_0, x_1, x_2] = \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0}$$

$$f[x_1, x_2] = \frac{f(x_2) - f(x_1)}{x_2 - x_1} = \frac{0.4771 - 0.3010}{3-2} = 0.1761$$

Therefore,

$$a_2 = \frac{0.1761 - 0.3010}{3-1} = -0.06245$$

$$\begin{aligned} p_2(x) &= a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) \\ &= 0 + 0.3010(x-1) + (-0.06245)(x-1)(x-2) \\ p_2(2.5) &= 0.3010 \times 1.5 - (0.06245)(1.5)(0.5) \\ &= 0.4515 - 0.0468 \\ &= 0.4047 \end{aligned}$$

Note that, in Example 9.6, had we used a linear polynomial, we would have obtained the result as follows:

$$p_1(x) = a_0 + a_1(x - x_0)$$

$$p_1(2.5) = 0 + 0.3010\,(1.5) = 0.4515$$

This shows that $p_2$ (2.5) is obtained by simply adding a correction factor due to third data point. That is

$$\begin{aligned} p_2(x) &= p_1(x) + a_2(x - x_0)(x - x_1) \\ &= p_1(x) + \Delta_2 \end{aligned}$$

If we want to improve the results further, we can apply further correction by adding another data point. That is

$$p_3(x) = p_2(x) + \Delta_3$$

where

$$\Delta_3 = a_3(x - x_0)(x - x_1)(x - x_2)$$

This shows that the Newton interpolation formula provides a very convenient form for interpolation at an increasing number of interpolation points. Newton formula can be expressed recursively as follows:

$$\boxed{p_{k+1}(x) = p_k(x) + f[x_0, \dots x_{k+1}]\,\phi_k(x)(x - x_k)} \qquad (9.17)$$

where $\quad p_k(x) = f[x_0, \dots x_i]\,\phi_i(x) = \sum_{i=0}^{k} a_i\,\phi_i(x)$

and $\quad \phi_i(x) = (x - x_0)(x - x_1) \dots (x - x_{i-1})$

## 9.6 DIVIDED DIFFERENCE TABLE

We have seen that the coefficients of Newton divided difference interpolation polynomial are evaluated using the divided differences at the interpolating points. We have also seen that a higher-order divided difference is obtained using the lower-order differences. Finally, the first-order divided differences use the given interpolating points (i.e., $x_k$ and $f_k$ values). For example, consider the second-order divided difference

$$a_2 = f[x_0, x_1, x_2]$$

$$= \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0}$$

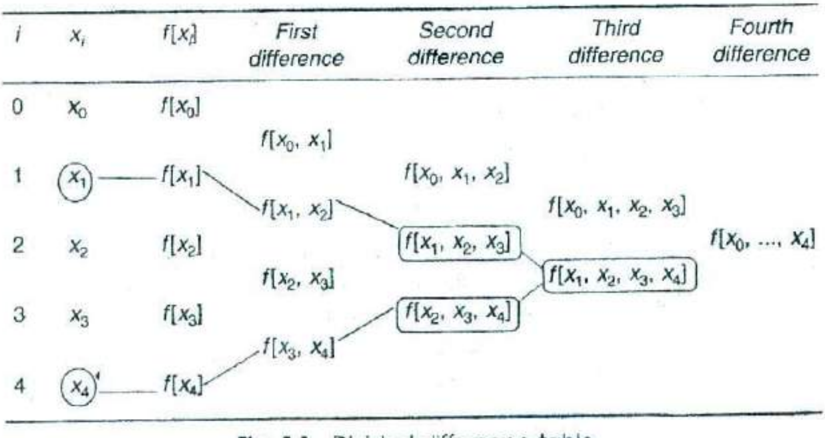
where $f[x_1, x_2]$ and $f[x_0, x_1]$ are first-order divided differences and are given by

$$f[x_0, x_1] = \frac{f(x_1) - f(x_0)}{x_1 - x_0} = \frac{f_1 - f_0}{x_1 - x_0}$$

$$f[x_1, x_2] = \frac{f(x_2) - f(x_1)}{x_2 - x_1} = \frac{f_2 - f_1}{x_2 - x_1}$$

This shows that, given the interpolating points, we can obtain recursively a higher-order divided difference, starting from the first-order differences. While this can be conveniently implemented in a computer, we can generate a *divided difference table* for manual computing. A divided difference table for five data points is shown in Fig. 9.3. A particular entry in the table is obtained as follows:

$$f[x_1, x_2, x_3, x_4] = \frac{f[x_2, x_3, x_4] - f[x_1, x_2, x_3]}{x_4 - x_1}$$



| $i$ | $x_i$ | $f[x_i]$ | First difference | Second difference | Third difference | Fourth difference |
|---|---|---|---|---|---|---|
| 0 | $x_0$ | $f[x_0]$ | | | | |
| | | | $f[x_0, x_1]$ | | | |
| 1 | $x_1$ | $f[x_1]$ | | $f[x_0, x_1, x_2]$ | | |
| | | | $f[x_1, x_2]$ | | $f[x_0, x_1, x_2, x_3]$ | |
| 2 | $x_2$ | $f[x_2]$ | | $f[x_1, x_2, x_3]$ | | $f[x_0, ..., x_4]$ |
| | | | $f[x_2, x_3]$ | | $f[x_1, x_2, x_3, x_4]$ | |
| 3 | $x_3$ | $f[x_3]$ | | $f[x_2, x_3, x_4]$ | | |
| | | | $f[x_3, x_4]$ | | | |
| 4 | $x_4$ | $f[x_4]$ | | | | |

**Fig. 9.3** Divided difference table

Draw the two diagonals from the entry to be calculated through its neighbouring entries to the left. If these lines terminate at $f(x_i)$ and $f(x_j)$, then divide the difference of the neighbouring entries by the corresponding difference $x_j - x_i$. The result is the desired entry. This is illustrated in Fig. 9.3. for the entry $f[x_1, x_2, x_3, x_4]$.

When the table is completed, the entries at the top of each column represent the divided difference coefficients.

### Example 9.7

Given the following set of data points, obtain the table of divided differences. Use the table to estimate the value of $f(1.5)$.

| $i$ | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| $x_i$ | 1 | 2 | 3 | 4 | 5 |
| $f(x_i)$ | 0 | 7 | 26 | 63 | 124 |

The divided difference table is given below:

| $i$ | $x_i$ | $f(x_i)$ | *First difference* | *Second difference* | *Third difference* | *Fourth difference* |
|---|---|---|---|---|---|---|
| 0 | 1 | 0 | | | | |
| | | | 7 | | | |
| 1 | 2 | 7 | | 12 | | |
| | | | 19 | | 6 | |
| 2 | 3 | 26 | | 18 | | 0 |
| | | | 37 | | 6 | |
| 3 | 4 | 63 | | 24 | | |
| | | | 61 | | | |
| 4 | 5 | 124 | | | | |

The value of polynomial at $x = 1.5$ is computed as follows:

$$p_0(1.5) = 0$$
$$p_1(1.5) = 0 + 7(1.5 - 1) = 3.5$$
$$p_2(1.5) = 3.5 + 12(1.5 - 1)(1.5 - 2) = 0.5$$
$$p_3(1.5) = 0.5 + 6\,(1.5 - 1)(1.5 - 2)(1.5 - 3) = 2.25$$
$$p_4(1.5) = 2.25 + 0 = 2.25$$

The function value at $x = 1.5$ is 2.25

Note that $p_3(1.5) = p_4(1.5)$. This implies that correct results can be obtained using the third-order interpolation polynomial. It also illustrates that we can compute $f(1.5)$ in stages (recursively) using interpolation polynomials in increasing order. Computation is terminated when two consecutive estimates are approximately equal or their difference is within a specified limit.

It is clear that the computational effort required in adding one more data point to the estimation process is very much reduced due to the recursive nature of computation.

Let us have a close look at the divided difference table of Example 9.7. Notice the constant values under the column "third difference" and zero value under the column "fourth difference". Recall that the first divided difference is given by

$$f[x_0,x_1]=\frac{f(x_1)-f(x_0)}{x_1-x_0}$$

This is nothing but the finite divided difference approximation of the first derivative of the function. Similarly, $f[x_0, x_1, x_2]$ is the second derivative and so on. Since the third derivative is constant, the function $f(x)$ should be a third-degree polynomial. In fact, the function used in Example 9.7 is

$$f(x) = x^3 - 1$$

and therefore

$$\frac{d'''f}{dx} = 6$$

and the fourth derivative is zero.

## Program NEWINT

Program NEWINT constructs the Newton interpolation polynomial for a given set of data points and then computes the interpolation value at a specified value.

```
* ---------------------------------------------------------- *
      PROGRAM NEWINT
* ---------------------------------------------------------- *
* Main program                                                *
*   This program constructs the Newton interpolation         *
*   polynomial for a given set of data points and then       *
*   computes interpolation value at a specified value        *
* ---------------------------------------------------------- *
* Functions invoked                                           *
*    NIL                                                      *
* ---------------------------------------------------------- *
* Subroutines used                                            *
*    NIL                                                      *
* ---------------------------------------------------------- *
* Variables used                                              *
*   N  - Number of data points                                *
*   X  - Array of independent data points                     *
*   F  - Array of function values                             *
*   XP - Desired point for interpolation                      *
*     FP - Interpolation value at XP                       *
*     D  - Difference table                                *
*     A  - Array of coefficients of interpolation          *
*          polynomial                                      *
* -------------------------------------------------------- *
* Constants used                                           *
*     NIL                                                  *
* -------------------------------------------------------- *
      INTEGER N
      REAL XP,FP,SUM,PI,X,F,A,D
      DIMENSION X(10),F(10),A(10),D(10,10)

      WRITE(*,*) 'Input number of data points"
      READ(*,*) N
      WRITE(*,*)     'Input the values of X and F(x), ',
     +               'one set on each line'
      DO 10 I = 1,N
         READ(*,*) X(I), F(I)
10    CONTINUE

* Construct difference table D

      DO 20 I = 1,N
         D(I,1) = F(I)
20    CONTINUE

      DO 40 J = 2,N
         DO 30 I = 1, N-J+1
          D(I,J) = (D(I+1,J-1)-D(I,J-1))/(X(I+J-1)-X(I))
30       CONTINUE
40    CONTINUE

* Set the coefficients of interpolating polynomial

      DO 50 J = 1,N
         A(J) = D(1,J)
50    CONTINUE

* Compute interpolation value

      WRITE(*,*) 'Input XP where interpolation is
                  required'
      READ(*,*) XP

      SUM = A(1)
      DO 70 I = 2,N
         PI = 1.0
         DO 60 J = 1, I-1
          PI = PI *(XP-X(J))
```

```
60   CONTINUE
     SUM = SUM + A(I) * PI
70  CONTINUE

    FP = SUM

* Write results

    WRITE(*,*)
    WRITE(*,*) 'NEWTON INTERPOLATION'
    WRITE(*,*)
    WRITE(*,*) 'Interpolated Function Value'
    WRITE(*,*) 'at X = ', XP, ' is', FP
    WRITE(*,*)

    STOP
    END
* --------------- End of main NEWINT ------------------ *
```

***Test Run Results*** Let us use the same table values that were used for testing the program LAGRAN. Test run results are given below:

```
Input number of data points
3
Input the values of X and F(x), one set on each line
2 1.4142
3 1.7321
4 2.0
Input XP where interpolation is required
2.5

NEWTON INTERPOLATION

Interpolated Function Value
at X = 2.5000000 is 1.5794000

Stop - Program terminated.
```

## 9.7 INTERPOLATION WITH EQUIDISTANT POINTS

In this section, we consider a particular case where the function values are given at a sequence of equally spaced points. Most of the engineering and scientific tables are available in this form. We often use such tables to estimate the value at a non-tabular point. Let us assume that

$$x_k = x_0 + kh$$

where $x_0$ is the reference point and $h$ is the step size. The integer $k$ may take either positive or negative values depending on the position of the reference point in the table. We also assume that we are going to use *simple differences* rather than *divided differences*. For this purpose, we define the following:

The *first forward difference* $\Delta f_i$ is defined as

$$\Delta f_i = f_{i+1} - f_i$$

The *second forward difference* is defined as

$$\Delta^2 f_i = \Delta f_{i+1} - \Delta f_i$$

In general,

$$\boxed{\Delta^j f_i = \Delta^{j-i} f_{i+1} - \Delta^{j-1} f_i} \tag{9.18}$$

We can now express the *simple forward differences* in terms of the *divided differences*. We know that

$$f[x_0, x_1] = \frac{f(x_1) - f(x_0)}{x_1 - x_0} = \frac{f_1 - f_0}{h}$$

Therefore,

$$f_1 - f_0 = h\, f[x_0, x_1]$$

Then

$$\Delta f_0 = f_1 - f_0 = h\, f[x_0, x_1]$$

Similarly,

$$\Delta f_1 = h\, f[x_1, x_2]$$

Now,

$$\begin{aligned}\Delta^2 f_0 &= \Delta f_1 - \Delta f_0 \\ &= h\, f[x_1, x_2] - h\, f[x_0, x_1] \\ &= h\, \{f[x_1, x_2] - f[x_0, x_1]\} \\ &= h\, .\, 2h\, .\, f[x_0, x_1, x_2] \\ &= 2\, h^2 f[x_0, x_1, x_2]\end{aligned}$$

In general, by induction,

$$\Delta^j f_i = j!\, h^j f[x_i, x_{i+1}, \dots x_{i+j}]$$

Therefore,

$$f[x_0, x_1, \dots x_j] = \frac{\Delta^j f_0}{j!h^j}$$

Substituting this in the Newton's divided difference interpolation polynomial (Eq. (9.16)) we get,

$$\boxed{p_n(x) = \sum_{j=0}^{n} \frac{\Delta^j f_0}{j!\,h^j} \prod_{k=0}^{j-1} (x - x_k)} \tag{9.19}$$

Let us set

$$x = x_0 + sh \qquad \text{and} \qquad p_n(s) = p_n(x)$$

We know that

$$x_k = x_0 + kh$$

Thus we get

$$x - x_k = (s - k)h$$

Substituting this in Eq. (9.19), we get

$$p_n(s) = \sum_{j=0}^{n} \frac{\Delta^j f_0}{j!\,h^j} \prod_{k=0}^{j-1} (s-k)h$$

$$= \sum_{j=0}^{n} \frac{\Delta^j f_0}{j!\,h^j} [s(s-1)\ldots(s-j+1)]\,h^j$$

Thus,

$$\boxed{p_n(s) = \sum_{j=0}^{n} \binom{s}{j} \Delta^j f_0} \qquad (9.20)$$

where

$$\binom{s}{j} = \frac{s(s-1)\ldots(s-j+1)}{j!}$$

is the binomial coefficient. Equations (9.19) and (9.20) are known as *Gregory-Newton forward difference formula.*

## Forward Difference Table

The coefficients $\Delta^j f_i$ can be conveniently obtained from the *forward difference table* shown in Fig. 9.4. According to Eq. (9.18), each entry is merely the difference between the two diagonal entries immediately on its left. That is

$$\Delta^j f_i = \Delta^{j-1} f_{i+1} - \Delta^{j-1} f_i$$

The differences which appear on the top of each column correspond to the differences of equation (9.20).

| $x$ | $f$ | $\Delta f$ | $\Delta^2 f$ | $\Delta^3 f$ | $\Delta^4 f$ | $\Delta^5 f$ | $\Delta^6 f$ |
|---|---|---|---|---|---|---|---|
| $x_0$ | $f_0$ | | | | | | |
| | | $\Delta f_0$ | | | | | |
| $x_1$ | $f_1$ | | $\Delta^2 f_0$ | | | | |
| | | $\Delta f_1$ | | $\Delta^3 f_0$ | | | |
| $x_2$ | $f_2$ | | $\Delta^2 f_1$ | | $\Delta^4 f_0$ | | |
| | | $\Delta f_2$ | | $\Delta^3 f_1$ | | $\Delta^5 f_0$ | |
| $x_3$ | $f_3$ | | $\Delta^2 f_2$ | | $\Delta^4 f_1$ | | |
| | | $\Delta f_3$ | | $\Delta^3 f_2$ | | | |
| $x_4$ | $f_4$ | | $\Delta^2 f_3$ | | | | |
| | | $\Delta f_4$ | | | | | |
| $x_5$ | $f_5$ | | | | | | |

**Fig. 9.4** Forward difference table

As pointed out earlier, difference tables can be used not only to estimate the value of the function at a non-tabular point but can also be used to decide on the degree of the interpolating polynomial that is most appropriate to the given data points.

### Example 9.8

Estimate the value of sin $\theta$ at $\theta = 25°$ using the Newton-Gregory forward difference formula with the help of the following table.

| $\theta$ | 10 | 20 | 30 | 40 | 50 |
|---|---|---|---|---|---|
| sin $\theta$ | 0.1736 | 0.3420 | 0.5000 | 0.6428 | 0.7660 |

In order to use the Newton-Gregory forward difference formula, we need the values of $\Delta^j f_0$. These coefficients can be obtained from the difference table given below. The required coefficients are boldfaced.

| $\theta$ | sin $\theta$ | $\Delta f$ | $\Delta^2 f$ | $\Delta^3 f$ | $\Delta^4 f$ | $\Delta^5 f$ |
|---|---|---|---|---|---|---|
| 10 | **0.1736** | | | | | |
| | | **0.1684** | | | | |
| 20 | 0.3420 | | **−0.0104** | | | |
| | | 0.1580 | | **0.0048** | | |
| 30 | 0.5000 | | − 0.0152 | | **− 0.0004** | |
| | | 0.1428 | | 0.0044 | | |
| 40 | 0.6428 | | − 0.0196 | | | |
| | | 0.1232 | | | | |
| 50 | 0.7660 | | | | | |

$$x_0 = \theta_0 = 10$$

$$h = 10$$

Therefore,

$$s = \frac{x - x_0}{h} = \frac{25 - 10}{10} = 1.5$$

Using Eq. (9.20), we have

$$p_1(s) = 0.1736 + (1.5)\,(0.1684) = 0.4262$$

$$p_2(s) = 0.4262 + \frac{(1.5)\,(0.5)\,(-0.0104)}{2} = 0.4223$$

$$p_3(s) = 0.4223 + \frac{(1.5)\,(0.5)\,(-0.5)\,(0.0048)}{6} = 0.4220$$

$$p_4(s) = 0.4220 + \frac{(1.5)\,(0.5)\,(-0.5)\,(-1.5)\,(-0.0004)}{24} = 0.4220$$

Thus,

$$\sin 25 = 0.4220$$

which is accurate to four decimal places.

## Backward Difference Table

If the table is too long and if the required point is close to the end of the table, we can use another formula known as *Newton-Gregory backward difference formula*. Here, the reference point is $x_n$, instead of $x_0$. Therefore, we have

$$x = x_n + sh$$

$$x_k = x_n - kh$$

$$x - x_k = (s + k)h$$

Then, the Newton-Gregory backward difference formula is given by

$$p_n(s) = f_n + s\ \nabla f_n + \frac{s(s+1)}{2!}\ \nabla^2 f_n + \ldots$$

$$+ \frac{s(s+1)\ldots(s+n-1)}{n!}\nabla^n f_n \qquad (9.21)$$

For a given table of data, the backward difference table will be identical to the forward difference table. However, the reference point will be below the point for which the estimate is required. This implies that the value of $s$ will be negative for backward interpolation. The coefficients $\nabla j\ f_i$ can be obtained from the backward difference table shown in Fig. 9.5.

| $x$ | $f$ | $\nabla f$ | $\nabla^2 f$ | $\nabla^3 f$ | $\nabla^4 f$ | $\nabla^5 f$ | $\nabla^6 f$ |
|---|---|---|---|---|---|---|---|
| $x_0$ | $f_0$ | | | | | | |
| | | $\nabla f_1$ | | | | | |
| $x_1$ | $f_1$ | | $\nabla^2 f_2$ | | | | |
| | | $\nabla f_2$ | | $\nabla^3 f_3$ | | | |
| $x_2$ | $f_2$ | | $\nabla^2 f_3$ | | $\nabla^4 f_4$ | | |
| | | $\nabla f_3$ | | $\nabla^3 f_4$ | | $\nabla^5 f_5$ | |
| $x_3$ | $f_3$ | | $\nabla^2 f_4$ | | $\nabla^4 f_5$ | | |
| | | $\nabla f_4$ | | $\nabla^3 f_5$ | | | |
| $x_4$ | $f_4$ | | $\nabla^2 f_5$ | | | | |
| | | $\nabla f_5$ | | | | | |
| $x_5$ | $f_5$ | | | | | | |

Fig. 9.5 Backward difference table

### Example 9.9

Repeat the estimation of sin 25 in Example 9.8 using Newton's backward difference formula

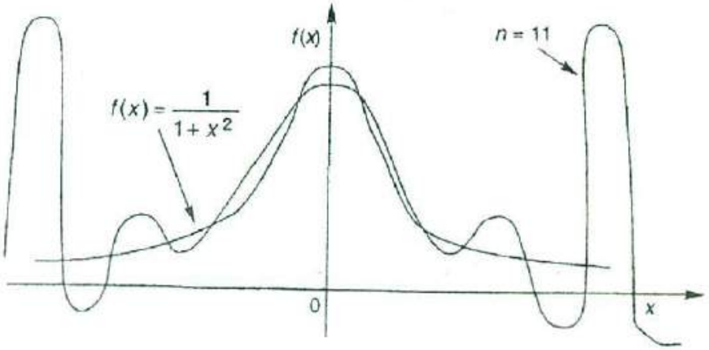
$$s = \frac{(x - x_n)}{h} = \frac{25 - 50}{10} = -2.5$$

Using Eq. (9.21), we get

$$\begin{aligned} p_4(2.5) &= 0.7660 + (-2.5)\,(0.1232) \\ &\quad + \frac{(-2.5)\,(-1.5)\,(-0.0196)}{2} \\ &\quad + \frac{(-2.5)\,(-1.5)\,(-0.5)\,(0.0044)}{6} \\ &\quad + \frac{(-2.5)\,(-1.5)\,(-0.5)\,(0.5)\,(-0.0004)}{24} \\ &= 0.4200 \end{aligned}$$

## 9.8 SPLINE INTERPOLATION

So far we have discussed how an interpolation polynomial of degree $n$ can be constructed and used given a set of values of functions. There are situations in which this approach is likely to face problems and produce incorrect estimates. This is because the interpolation takes a global rather than a local view of data. It has been proved that when $n$ is large compared to the order of the "true" function, the interpolation polynomial of degree $n$ does not provide accurate results at the ends of the range. This is illustrated in Fig. 9.6. Note that the interpolation polynomial contains undesirable maxima and minima between the data points. This only shows that increasing the order of polynomials does not necessarily increase the accuracy.



Ao3
A13

Fig. 9.6 Interpolation polynomial of degree 11 of the function $\frac{1}{1+x^2}$

One approach to overcome this problem is to divide the entire rang of points into subintervals and use local low-order polynomials to interpolate each subinterval. Such polynomials are called *piecewise polynomials*. Subintervals are usually taken as $[x_i, x_{i+1}]$, $i = 0, 1, \ldots n$ as illustrated in Fig. 9.7.

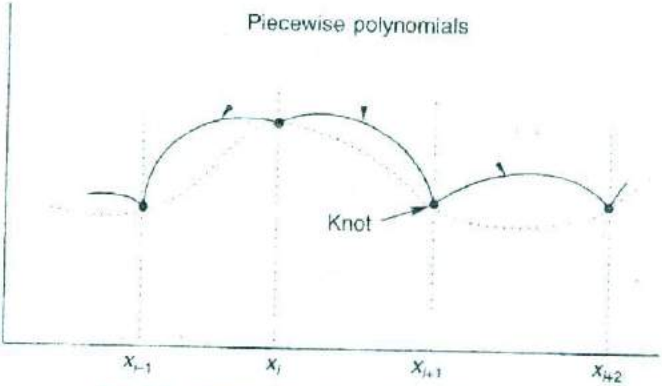


**Fig. 9.7** Piecewise polynomial interpolation

Notice that the piecewise polynomials shown in Fig. 9.7 exhibit discontinuity at the interpolating points (which connect these polynomials). It is possible to construct piecewise polynomials that prevent such discontinuities at the connecting points. Such piecewise polynomials are called *spline functions* (or simply *splines*). Spline functions, therefore, look smooth at the connecting points as shown in Fig. 9.8. The connecting points are called *knots* or *nodes* (because this is where the polynomial pieces are tied together).

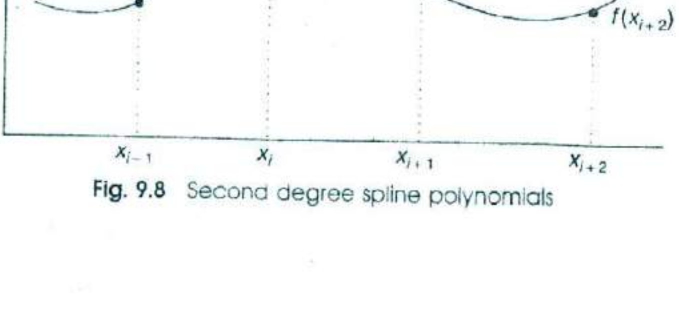


**Fig. 9.8** Second degree spline polynomials

A spline function $s(x)$ of degree $m$ must satisfy the following conditions:

1. $s(x)$ is a polynomial of degree *atmost* $m$ in each of the subintervals $[x_i, x_{i+1}]$, $i = 0, 1, \dots n$.
2. $s(x)$ and its derivatives of orders 1,2 ... $m - 1$ are continuous in the range $[x_0, x_n]$.

According to the first condition, each interval will have a different polynomial of degree $m$ or less. The set of all polynomials form a *spline interpolation polynomial*, if $s(x_i) = f_i$, for $i = 0, 1, \dots n$. The process of constructing such polynomials for a given set of function points is known as *spline interpolation*.

### Example 9.10

State whether the following piecewise polynomials are splines or not.

$$\text{(i)}\ f(x) = \begin{cases} x+1 & -1 \le x \le 0 \\ 2x+1 & 0 \le x \le 1 \\ 4-x & 1 \le x \le 2 \end{cases}$$

$$\text{(ii)}\ f(x) = \begin{cases} x^2+1 & 0 \le x \le 1 \\ 2x^2 & 1 \le x \le 2 \\ 5x-2 & 2 \le x \le 3 \end{cases}$$

$$\text{(iii)}\ f(x) = \begin{cases} x & 0 \le x \le 1 \\ x^2-x+1 & 1 \le x \le 2 \\ 3x-3 & 2 \le x \le 3 \end{cases}$$

---

*Case* $(i)$

Given,

$n = 4, \quad x_0 = -1, \quad x_1 = 0, \quad x_2 = 1, \quad x_3 = 2$

$$f_1(x) = x + 1$$
$$f_2(x) = 2x + 1$$
$$f_3(x) = 4 - x$$

Then,

$$f_1(x_1) = 0 + 1 = 1$$
$$f_2(x_1) = 2 \times 0 + 1 = 1$$
$$f_2(x_2) = 2 + 1 = 3$$
$$f_3(x_2) = 4 - 1 = 3$$

Note that

$$f_1(x_1) = f_2(x_1) \qquad \text{and} \qquad f_2(x_2) = f_3(x_2)$$

Therefore, the piecewise polynomials are continuous and $f(x)$ is a linear spline. Note that the first-derivative is not continuous and, therefore, $f(x)$ is not a second-degree spline.

*Case (ii)*

Given,

$n = 4, \quad x_0 = 0, \quad x_1 = 1, \quad x_2 = 2, \quad x_3 = 3$

$$f_1(x) = x^2 + 1, \qquad f_1'(x) = 2x$$
$$f_2(x) = 2x^2, \qquad f_2'(x) = 4x$$
$$f_3(x) = 5x - 2, \qquad f_3'(x) = 5$$

Then,

$$f_1(x_1) = 1 + 1 = 2, \qquad f_1'(x_1) = 2$$
$$f_2(x_1) = 2 \times 1 = 2, \qquad f_2'(x_1) = 4$$
$$f_2(x_2) = 2 \times 4 = 8, \qquad f_2'(x_2) = 8$$
$$f_3(x_2) = 5 \times 2 - 2 = 8, \qquad f_3'(x_2) = 5$$

Polynomials are continuous but their derivatives are not. Therefore, $f(x)$ is not a spline.

*Case (iii)*

Given,

$n = 4, \quad x_0 = 0, \quad x_1 = 1, \quad x_2 = 2, \quad x_3 = 3$

$$f_1(x) = x, \qquad f_1'(x) = 1, \qquad f_1''(x) = 0$$
$$f_2(x) = x^2 - x + 1, \qquad f_2'(x) = 2x - 1, \qquad f_2''(x) = 2$$
$$f_3(x) = 3x - 3, \qquad f_3'(x) = 3, \qquad f_3''(x) = 0$$

Then,

$$f_1(x_1) = 1 \qquad f_1'(x_1) = 1$$
$$f_2(x_1) = 1 \qquad f_2'(x_1) = 1$$
$$f_2(x_2) = 3 \qquad f_2'(x_2) = 3$$
$$f_3(x_2) = 3 \qquad f_3'(x_2) = 3$$

Since both the polynomials and their first derivatives are continuous in the given interval, $f(x)$ is a second-degree spline. Note that the second derivatives are not continuous.

## Cubic Splines

The concept of splines originated from the mechanical drafting tool called "spline" used by designers for drawing smooth curves. It is a slender flexible bar made of wood or some other elastic material. These curves resemble cubic curves and hence the name "cubic spline" has been given to the piecewise cubic interpolating polynomials. Cubic splines are popular because of their ability to interpolate data with smooth curves. It is believed that a cubic polynomial spline always appears smooth to the eyes.

We consider here the construction of a cubic spline function which would interpolate the points $(x_0, f_0)$, $(x_1, f_1)$, ... $(x_n, f_n)$. The cubic spline $s(x)$ consists of $(n - 1)$ cubics corresponding to $(n - 1)$ subintervals. If we denote such cubic by $s_i(x)$, then

$$s(x) = s_i(x), \quad i = 1,2,...n$$

As pointed out earlier, these cubics must satisfy the following conditions:

1. $s(x)$ must interpolate $f$ at all the points $x_0, x_1, ... x_n$, i.e, for each $i$
$$s(x_i) = f_i \tag{9.22}$$
2. The function values must be equal at all the interior knots
$$s_i(x_i) = s_{i+1}(x_i) \tag{9.23}$$
3. The first derivatives at the interior knots must be equal
$$s_i'(x_i) = s_{i+1}'(x_i) \tag{9.24}$$
4. The second derivatives at the interior knots must be equal
$$s_i''(x_i) = s_{i+1}''(x_i) \tag{9.25}$$
5. The second derivative at the end points are zero
$$s''(x_0) = s''(x_n) = 0$$

*Step 1*

Let us first consider the second derivatives. Since $s_i\ (x)$ is a cubic function, its second derivative $s_i''(x)$ is a straight line. This straight line can be represented by a first-order Lagrange interpolating polynomial. Since the line passes through the points $(x_i,\ s_i''(x_i))$ and $(x_{i-1},\ s_i''(x_{i-1}))$, we have,

$$s_i''(x) = s_i''(x_{i-1})\ \frac{x - x_i}{x_{i-1} - x_i} + s_i''(x_i)\frac{x - x_{i-1}}{x_i - x_{i-1}} \tag{9.26}$$

The unknowns $s_i''(x_{i-1})$ and $s_i''(x_i)$ are to be determined. For the sake of simplicity, let us denote

$$s_i''(x_{i-1}) = a_{i-1} \qquad \text{and} \qquad s_i''(x_i) = a_i$$
$$x - x_i = u_i$$
$$x_i - x_{i-1} = h_i = u_{i-1} - u_i$$

Then, Eq. (9.26) becomes

$$s_i''(x) = a_{i-1}\ \frac{u_i}{-h_i} + a_i\ \frac{u_{i-1}}{h_i}$$

$$= \frac{a_i u_{i-1} - a_{i-1} u_i}{h_i} \tag{9.27}$$

*Step 2*

Now we can obtain $s_i(x)$ by integrating Eq. (9.27) twice. Thus

$$s_i(x) = \frac{a_i u_{i-1}^3 - a_{i-1} u_i^3}{6h_i} + C_1 x + C_2 \tag{9.28}$$

where $C_1$ and $C_2$ are constants of integration [observe that $du_i/d_x = 1$ and, therefore, differentiation and integration with respect to $x$ and with respect to $u_i$ will be equivalent]. The linear part $C_1 x + C_2$ can be expressed as

$$b_1\,(x - x_{i-1}) + b_2\,(x - x_i)$$

with suitable choice of $b_1$ and $b_2$.

Therefore,

$$\begin{aligned} C_1\,x + C_2 &= b_1\,(x - x_{i-1}) + b_2\,(x - x_i) \\ &= b_1\,u_{i-1} + b_2\,u_i \end{aligned}$$

Then, Eq. (9.28) becomes,

$$s_i(x) = \frac{a_i u_{i-1}^3 - a_{i-1} u_i^3}{6h_i} + b_1 u_{i-1} + b_2 u_i$$

*Step 3*

Now, we must determine the coefficients $b_1$ and $b_2$. We know that, by condition 1,

$$s(x_i) = f_i \qquad \text{and} \qquad s(x_{i-1}) = f_{i-1}$$

At $x = x_i$,

$$u_i = 0,\ u_{i-1} = h_i$$

$$f_i = \frac{a_i h_i^2}{6} + b_1 h_i$$

Similarly, at $x = x_{i-1}$

$$u_{i-1} = 0, \qquad u_i = -h_i$$

and therefore

$$f_{i-1} = \frac{a_{i-1} h_i^2}{6} - b_2 h_i$$

Thus, we get

$$b_1 = \frac{f_i}{h_i} - \frac{a_i h_i}{6} \tag{9.29a}$$

$$b_2 = -\frac{f_{i-1}}{h_i} + \frac{a_{i-1} h_i}{6} \tag{9.29b}$$

Substituting for $b_1$ and $b_2$ in Eq. (9.29) and after rearrangement of terms, we get

$$s_i(x) = \frac{a_{i-1}}{6h_i}(h_i^2 u_i - u_i^3) + \frac{a_i}{6h_i}(u_{i-1}^3 - h_i^2 u_{i-1})$$

$$+ \frac{1}{h_i}(f_i u_{i-1} - f_{i-1} u_i) \tag{9.30}$$

Note that Eq. (9.30) has only two unknowns, $a_{i-1}$ and $a_i$.

*Step 4*

The final step is to evaluate these constants. This can be done by invoking the condition

$$s_i'(x_i) = s_{i+1}'(x_i)$$

Differentiating Eq. (9.30) we get

$$s_i'(x) = \frac{a_{i-1}}{6h_i}(h_i^2 - 3u_i^2)$$

$$+ \frac{a_i}{6h_i}(3u_{i-1}^2 - h_i^2)$$

$$+ \frac{1}{h_i}(f_i - f_{i-1})$$

Ashish

Setting $x = x_i$,

$$s_i'(x_i) = \frac{a_{i-1}h_i}{6} + \frac{a_i h_i}{3} + \frac{f_i - f_{i-1}}{h_i}$$

NUB

Similarly,

$$s_{i+1}'(x_i) = -\frac{a_i h_{i+1}}{3} - \frac{a_{i+1}h_{i+1}}{6} + \frac{f_{i+1} - f_i}{h_{i+1}}$$

Since

$$s_i'(x_i) = s_{i+1}'(x_i)$$

We have

$$\boxed{h_i a_{i-1} + 2(h_i + h_{i+1})a_i + h_{i+1}a_{i+1} = 6\left[\frac{f_{i+1} - f_i}{h_{i+1}} - \frac{f_i - f_{i-1}}{h_i}\right]} \tag{9.31}$$

Equation (9.31), when written for all interior knots ($i = 1, \ldots n - 1$), we get $n - 1$ simultaneous equations containing $n + 1$ unknowns ($a_0, a_1, \ldots a_n$). Now, applying the condition that the second derivatives at the end points are zero, we get

$$a_0 = a_n = 0$$

Thus, we have $n - 1$ equations with $n - 1$ unknowns which can be easily solved.

> *Note*
>
> The cubic splines with zero second derivatives at the end points are called the *natural cubic splines*. This is because the splines are assumed to take their natural straight line shape outside the intervals of approximations.

The system of $n - 1$ equations contained in Eq. (9.31) can be expressed as

$$\begin{bmatrix} 2(h_1+h_2) & h_2 & 0 & \cdots & 0 & 0 & 0 \\ h_2 & 2(h_2+h_3) & h_3 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & h_{n-2} & 2(h_{n-2}+h_{n-1}) & h_{n-1} \\ 0 & 0 & 0 & \cdots & 0 & h_{n-1} & 2(h_{n-1}+h_n) \end{bmatrix}$$

$$\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} D_1 \\ D_2 \\ \vdots \\ \vdots \\ D_{n-1} \end{bmatrix} \qquad (9.32)$$

where

$$D_i = 6\left[\frac{f_{i+1} - f_i}{h_{i+1}} - \frac{f_i - f_{i-1}}{h_i}\right]$$

$$h_i = x_i - x_{i-1}$$

$$i = 1, 2, \ldots n - 1$$

### Example 9.11

Given the data points

| $i$ | 0 | 1 | 2 |
|---|---|---|---|
| $x_i$ | 4 | 9 | 16 |
| $f_i$ | 2 | 3 | 4 |

estimate the function value $f$ at $x = 7$ using cubic splines.

$$h_1 = x_1 - x_0 = 9 - 4 = 5$$

$$h_2 = x_2 - x_1 = 16 - 9 = 7$$

$$f_0 = 2, \qquad f_1 = 3, \qquad f_2 = 4$$

From Eq. (9.31), we have, for $i = 1$,

$$h_1 a_0 + 2(h_1 + h_2)a_1 + h_2 a_2 = 6\left[\frac{f_2 - f_1}{h_2} - \frac{f_1 - f_0}{h_1}\right]$$

We know that $a_0 = a_2 = 0$. Thus,

$$2(5 + 7)a_1 = 6\left[\frac{1}{7} - \frac{1}{5}\right]$$

Therefore,

$$a_1 = \frac{(6)(-2)}{(35)(24)} = -0.0143$$

Since $n = 3$, there are two cubic splines, namely,

$$s_1(x) \qquad x_0 \le x \le x_1$$
$$s_2(x) \qquad x_1 \le x \le x_2$$

The target point $x = 7$ is in the domain of $s_1(x)$ and, therefore, we need to use only $s_1(x)$ for estimation.

From Eq. (9.30)

$$s_1(x) = \frac{a_1(u_0^3 - h_1^2 u_0)}{6h_1} + \frac{1}{h_1}(f_1 u_0 - f_0 u_1)$$

$$u_0 = x - x_0 \qquad \text{and} \qquad u_1 = x - x_1$$

Upon substitution of specific values,

$$s_1(7) = -\frac{0.0143}{6 \times 5}\,[(7-4)^3 - 5^2\,(7-4)]$$

$$= +\frac{1}{5}\,[3(7-4) - 2(7-9)]$$

$$= 2.6229$$

## Algorithm

Note that Eq. (9.32) form a *tridiagonal* system which is relatively simple to solve using Gauss elimination method. A detailed solution procedure to evaluate spline functions is given in Algorithm 9.1

### Natural cubic spline

1. Provide input data.
2. Compute step lengths and form function differences.
3. Obtain the coefficients of the tridiagonal matrix.
4. Compute the right-hand side (*D* array) of the system.
5. Compute the elements $a_i$ using Gauss elimination method.

*(Contd.)*

*(Contd.)*

6. Evaluate the coefficients of natural cubic splines
7. Evaluate the spline function at the point of interest.
8. Print results.

**Algorithm 9.1**

## Program SPLINE

Natural cubic splines interpolation uses Gauss elimination method to implement its algorithm. Program SPLINE, therefore, calls for the help of GAUSS subprogram to compute the array of second derivatives.

```
* ---------------------------------------------------------- *
      PROGRAM SPLINE
* ---------------------------------------------------------- *
* Main program                                               *
*     This program computes the interpolation value at       *
*     a specified value, given a set of table points,        *
*     using the natural cubic spline interpolation           *
* ---------------------------------------------------------- *
* Functions invoked                                          *
*     NIL                                                    *
* ---------------------------------------------------------- *
* Subroutines used                                           *
*     GAUSS                                                  *
* ---------------------------------------------------------- *
* Variables used                                             *
*       N  - Number of data points.                          *
*       X  - N by 1 array of data points.                    *
*       F  - N by 1 array of function values                 *
*       XP - Point at which interpolation is required        *
*       FP - Interpolation value at XP                       *
*       A  - Array of second derivatives (N-2 by 1)          *
*       D  - Array representing right side of (9.32)         *
*            (N-2 by 1)                                      *
*       C  - Matrix (N-2 by N-2) representing the            *
*            coefficients of second derivatives              *
*       H  - Array of distances between data points          *
*            (h(i) = x(i) - x(i-1))                          *
*       DF - Array of differences of functions               *
* ---------------------------------------------------------- *
* Constants used                                             *
*       MAX - Maximum number of table points permitted       *
* ---------------------------------------------------------- *
```

```
      INTEGER N,MAX
      REAL XP,FP,X,F,A,D,C,H,DF,U
      PARAMETER (MAX=10)
      DIMENSION X(MAX),F(MAX),A(MAX),D(MAX),C(MAX,MAX),
     +          H(MAX),DF(MAX),U(MAX)
* Read input data
      WRITE(*,*) 'Input number of data points n'
      READ(*,*) N
      WRITE(*,*) 'Input data points X(I) and function'
      WRITE(*,*) 'values F(I), one set in each line'
      DO 5 I = 1,N
         READ(*,*) X(I), F(I)
5     CONTINUE
      WRITE(*,*) 'Input XP'
      READ(*,*) XP
* Compute distances between data points
* and function differences
      DO 10 I = 2,N
         H(I) = X(I) - X(I-1)
         DF(I) = F(I) - F(I-1)
10    CONTINUE
* Initialise C matrix
      DO 30 I = 2,N-1
         DO 20 J = 2, N-1
          C(I,J) = 0.0
20       CONTINUE
30    CONTINUE
* Compute diagonal elements of C
      DO 40 I = 2,N-1
         C(I,I) = 2.0 * (H(I)+H(I+1))
40    CONTINUE
* Compute off_diagonal elements of C
      DO 50 I = 3,N-1
         C(I-1,I) = H(I)
         C(I,I-1) = H(I)
50    CONTINUE
*  Compute elements of D array
      DO 60 I = 2,N-1
         D(I) = (DF(I+1)/H(I+1) - DF(I)/H(I)) * 6.0
```

```
60   CONTINUE

* Compute elements of A using Gaussian elimination
* Change array subscripts from 2 to n-1 to 1 to n-1
* before calling GAUSS

     M = N-2
     DO 80 I = 1,M
         D(I) = D(I+1)
         DO 70 J = 1,M
           C(I,J) = C(I+1,J+1)
70       CONTINUE
80   CONTINUE

     CALL GAUSS(M,C,D,A)

* Compute the coefficients of natural cubic spline

     DO 90 I = N-1,2,-1
         A(I) = A(I-1)
90   CONTINUE
     A(1) = 0.0
     A(N) = 0.0

* Locatc the domain of XP

     I = 2
100  IF( XP .LE. X(I) ) GO TO 110
     I = I+1
     GO TO 100

* Compute interpolation value at XP
* Use equation (9.30)

110  U(I-1) = XP - X(I-1)
     U(I) = XP - X(I)
     Q1 = H(I)**2 * U(I) - U(I)**3
     Q2 = U(I-1)**3 - H(I)**2 * U(I-1)
     Q3 = F(I) * U(I-1) - F(I-1) * U(I)
     FP = (A(I-1) * Q1 + A(I) * Q2)/(6.0 * H(I))
            + Q3/H(I)

* Write results

     WRITE(*,*)
     WRITE(*,*) 'SPLINE INTERPOLATION'
     WRITE(*,*)
     WRITE(*,*) 'Interpolation value =',FP
     WRITE(*,*)

     STOP
     END
* --------------- End of main SPLINE ------------------ *
* ------------------------------------------------------------ *
*    SUBROUTINE GAUSS(N,A,B,X)                                   *
* ------------------------------------------------------------ *
* Subroutine                                                     *
*    This subroutine solves a set of n linear                   *
*    equations using Gauss elimination method                   *
* ------------------------------------------------------------ *
* Arguments                                                      *
* Input                                                          *
*    N - Number of equations                                     *
*    A - Matrix of coefficients                                  *
*    B - Right side vector                                       *
* Output                                                         *
*    X - Solution vector                                         *
* ------------------------------------------------------------ *
* Local Variables                                                *
*    PIVOT, FACTOR, SUM                                          *
* ------------------------------------------------------------ *
* Functions invoked                                              *
*    NIL                                                         *
* ------------------------------------------------------------ *
* Subroutines called                                             *
*    NIL                                                         *
* ------------------------------------------------------------ *

      INTEGER N
      REAL A,B,X,PIVOT,FACTOR,SUM
      DIMENSION A(10,10), B(10), X(10)
* ---------------- Elimination begins ------------------------- *

      DO 33 K = 1, N-1
         PIVOT = A(K,K)
         DO 22 I = K+1, N
           FACTOR = A(I,K)/PIVOT
           DO 11 J = K+1, N
              A(I,J) = A(I,J) - FACTOR * A(K,J)
11         CONTINUE
           B(I) = B(I) - FACTOR * B(K)
22       CONTINUE
33    CONTINUE
* ------------- Back substitution begins ---------------------- *

      X(N) = B(N)/A(N,N)
      DO 55 K = N-1,1,-1
         SUM = 0
         DO 44 J = K+1,N
```

```
                SUM = SUM + A(K,J) * X(J)
44          CONTINUE
            X(K) = (B(K) - SUM)/A(K,K)
55      CONTINUE

        RETURN
        END
* ------------ End of subroutine GAUSS --------------- *
```

***Test Run Results*** Program SPLINE was tested using the table of data points given in Example 9.11.

Results are given below:

```
Input number of data points n
3
Input data points, X(I) and function
values F(I), one set in each line
4 2
9 3
16 4
Input XP
7

SPLINE INTERPOLATION

Interpolation Value = 2.6228570

Stop - Program terminated.
```

## Equidistant Knots

Most often the knots are equally spaced. This would simplify the solution considerably. If the knots are equally spaced,

$$h_1 = h_2 = \ldots = h_n = h.$$

Substituting this in equations (9.11) and dividing throughout by $h$, we get

$$\begin{bmatrix} 4 & 1 & 0 & \cdots & 0 & 0 & 0 \\ 1 & 4 & 1 & \cdots & \vdots & \vdots & \vdots \\ 0 & 1 & 4 & 1 & \vdots & \vdots & \vdots \\ \vdots & \cdots & \cdots & \cdots & \vdots & \vdots & \vdots \\ \vdots & \cdots & \cdots & \cdots & 4 & 1 & 0 \\ 0 & \cdots & \cdots & \cdots & 1 & 4 & 1 \\ 0 & 0 & 0 & \cdots & 0 & 0 & 4 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \\ \vdots \\ \vdots \\ \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \\ \vdots \\ \vdots \\ \\ d_{n-1} \end{bmatrix} \tag{9.33}$$

where

$$d_i = \frac{D_i}{h} = \frac{6(f_{i+1} - 2f_i + f_{i-1})}{h^2}$$

$$= \frac{6}{h^2}\Delta^2 f_{i-1}$$

$$= 12\,f[x_{i-1}, x_i, x_{i+1}] \qquad i = 1,2, \ldots, n-1$$

### Example 9.12

Given the table of values

| $i$ | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| $x_i$ | 1 | 2 | 3 | 4 |
| $f(x_i)$ | 0.5 | 0.3333 | 0.25 | 0.20 |

estimate the value of $f(2.5)$ using cubic spline functions

The points are equally spaced and therefore

$$h_1 = h_2 = h_3 = 1$$

Since $n = 4$, we have three intervals and three cubics and, therefore, only $a_1$ and $a_2$ are to be determined. From Eq. (9.33), we have

$$\begin{bmatrix} 4 & 1 \\ 1 & 4 \end{bmatrix}\begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \end{bmatrix}$$

$$d_1 = \frac{6}{h^2}\,(f_2 - 2f_1 + f_0)$$

$$= 6\,(0.25 - 2 \times 0.3333 + 0.5)$$

$$= 0.5004$$

$$d_2 = \frac{6}{h^2}\,(f_3 - 2f_2 + f_1)$$

$$= 6(0.2 - 2 \times 0.25 + 0.3333)$$

$$= 0.1998$$

Solving for $a_1$ and $a_2$

$$a_1 = \frac{d_1 \times 4 - d_2 \times 1}{15}$$

$$= \frac{0.5004 \times 4 - 0.1998}{15} = 0.1201$$

$$a_2 = \frac{d_2 \times 4 - d_1 \times 1}{15}$$

$$= \frac{0.1998 \times 4 - 0.5004}{15} = 0.0199$$

The target point $x = 2.5$ is in the domain of $s_2(x)$. Using **Eq. (9.30)**,

$$s_2(x) = \frac{a_1}{6}(u_2 - u_2^3) + \frac{a_2}{6}(u_1^3 - u_1) + (f_2 u_1 - f_1 u_2)$$

$$= \frac{a_1}{6}[(x - x_2) - (x - x_2)^3] + \frac{a_2}{6}[(x - x_1)^3$$

$$- (x - x_1)] + [f_2(x - x_1) - f_1(x - x_2)]$$

Upon substitution of values, we get

$$s_2(2.5) = \frac{0.1201}{6}[(2.5 - 3) - (2.5 - 3)^3]$$

$$+ \frac{0.0199}{6}[(2.5 - 2)^3 - (2.5 - 2)]$$

$$+ (0.25)\,(2.5 - 2) - 0.3333\,(2.5 - 3)$$

$$= -0.0075 - 0.0012 + 0.125 + 0.1667$$

$$= 0.2829$$

## CHEBYSHEV INTERPOLATION POLYNOMIAL

Recall that the truncation error in approximating a function $f(x)$ by an interpolating polynomial $p_n(x)$ with interpolation points $x_i$, $i = 0,1,...n$ is

$$f(x) - p_n(x) = w_n(x)\frac{f^{(n+1)}(\theta)}{(n+1)!}$$

where

$$w_n(x) = (x - x_0)\,(x - x_1)\,...\,(x - x_n)$$

and $\theta$ is some point in the interval of interest. One of the goals while applying an interpolation polynomial is to minimise the truncation error. Since $f^{(n+1)}(\theta)$ is not in our control, we can try to minimise the absolute value of $w_n(x)$. This can be done by choosing a proper set of interpolating points $x_i$ in the given interval $(a, b)$.

### Chebyshev Points

The Russian mathematician Chebyshev showed that the error bound is minimum when the interpolation points are chosen as follows:

$$x_k = \frac{a+b}{2} + \frac{a-b}{2}\cos\left[\frac{2k+1}{2(n+1)}\pi\right] \qquad k = 0, 1, ..., n \qquad (9.34)$$

These values are called *Chebyshev nodes* (or *points*). We can evaluate function values at these points. That is

$$f_k = f(x_k)$$

Now, we can apply the Lagrange interpolation method to the Chebyshev points and the corresponding function values to obtain an interpolation polynomial known as *Lagrange - Chebyshev interpolation polynomial.*

## Chebyshev Polynomials

Another approach to construct the interpolation polynomial $p_n(x)$ is to use Chebyshev polynomials as basis polynomials. That is

$$p_n(x) = C_0 t_0(t) + C_1 T_1(t) + \dots + C_n T_n(t)$$

$$= \sum_{i=0}^{n} C_i T_i(t) \tag{9.35}$$

where $T_i(t)$ is the *Chebyshev basis polynomial* of order $i$ in $t$ and $C_i$ the Chebyshev coefficient. Equation (9.35) is known as Chebyshev interpolation polynomial. Chebyshev polynomial $T_i(t)$ is given by

$$T_0(t) = 1$$
$$T_1(t) = t$$
$$T_i(t) = 2t\, T_{i-1}(t) - T_{i-2}(t) \qquad k = 2, \dots, n$$

$C_i$ are computed as follows:

$$C_0 = \frac{1}{n+1} \sum_{k=0}^{n} f(x_k)\, T_0\,(t_k) = \frac{1}{n+1} \sum_{k=0}^{n} f(x_k)$$

$$C_j = \frac{2}{n+1} \sum_{k=0}^{n} f(x_k)\, T_j\,(t_k)$$

where

$$T_j(t_k) = \cos\left[ j\, \frac{(2k+1)\pi}{2(n+1)} \right]$$

Therefore

$$C_j = \frac{2}{n+1} \sum_{k=0}^{n} f(x_k) \cos\left[ j\, \frac{(2k+1)\pi}{2(n+1)} \right] j = 1, 2, \dots, n$$

Evaluation of $p_n\,(x)$, given $x$:

$$t = \frac{x - (b+a)/2}{(b-a)/2}$$

$$p_n(x) = \sum_{i=0}^{n} C_i T_i\,(t)$$

## 9.10 SUMMARY

In this chapter, we discussed various methods for constructing interpolation polynomials for tables of well-defined functions. They include:

- Lagrange interpolation
- Newton's interpolation
- Newton-Gregory forward interpolation
- Spline interpolation

To facilitate the construction of interpolation functions, we presented different forms of polynomials that included

- power form
- shifted power form
- Newton form

We have also discussed how to build different types of difference tables and how to use them for estimating function values at any point. Finally, we considered how Chebyshev points and Chebyshev polynomials may be used to minimise the truncation error.

We have given computer programs and test results for the following methods:

- Lagrange interpolation
- Newton's interpolation
- Spline interpolation

## Key Terms

*Approximating functions*
*Backward difference*
*Central cubic spline*
*Central difference*
*Chebyshev basis polynomial*
*Chebyshev interpolation*
*Chebyshev points*
*Chebyshev polynomial*
*Cubic spline*
*Curve fitting*
*Divided difference table*
*Divided differences*
*Forward difference*
*Interpolation*
*Interpolation function*
*Interpolation polynomial*
*Knots*
*Lagrange basis polynomial*
*Lagrange interpolation*
*Lagrange interpolation polynomial*
*Leat-squares polynomials*
*Leat-squares regression*
*Linear interpolation*
*Natural cubic spline*
*Newton form*
*Newton interpolation polynomial*
*Newton's interpolation*
*Newton-Gregory formula*
*Newton-Gregory interpolation*
*Nodes*
*Piecewise polynomial*
*Power form*
*Shifted power form*
*Simple difference*
*Spline function*
*Spline interpolation*
*Spline interpolation polynomial*
*Splines*
*Taylor expansion*
*Tridiagonal system*

## REVIEW QUESTIONS

1. What is curve fitting? What is the need for such an exercise?
2. What is interpolation?
3. What are the methods available for interpolation?
4. Discuss the possible sources of errors in interpolation?
5. What is interpolation function?
6. List, with examples, different forms of polynomials that could be used for constructing interpolation functions.
7. Given two points $(x_1, y_1)$ and $(x_2, y_2)$, state the linear interpolation formula in terms of these points.
8. Given a set of $n + 1$ points, state the general form of $n$th degree Lagrange interpolation polynomial.
9. What is the computational effort required in using Lagrange polynomial?
10. What is the major pitfall of using Lagrange polynomial?
11. What are divided differences?
12. State the second order Newton's divided difference interpolation polynomial.
13. How is the Newton's interpolation formula better than Lagrange formula?
14. What is a divided difference table? How is it useful?
15. Construct a divided difference table for four data points.
16. Entries under a particular column in a divided difference table are constants. What does it indicate?
17. Distinguish between the simple difference and divided difference.
18. What is the difference between the forward difference table and backward difference table?
19. Look at Examples 9.8 and 9.9. Answers are different. Why?
20. What are piecewise polynomials?
21. What are spline functions?
22. What is spline interpolation?
23. What are cubic splines?
24. State the conditions for a spline to be cubic.
25. What are natural cubic splines?
26. What is tridiagonal system?
27. State the contribution of Russian mathematician Chebyshev in minimizing the truncation error in interpolation.

## REVIEW EXERCISES

1. Construct the power form of the straight line $p(x)$ which takes on the values

   $p(200) = 1/3$

$p(202) = -2/3$

using four-digit floating-point arithmetic.

2. Solve the problem in Exercise 1 using the shifted-power form and compare the results.
3. Find the linear interpolation polynomial for each of the following pairs of points:
   (a) (0, 1) and (1, 3)
   (b) (–2, 3) and (7, 12)
4. Find the quadratic interpolating polynomial for each of the following set of points:
   (a) (–1, 1), (0, 1) and (1, 3)
   (b) (0, –1), (1, 0) and (2, 9)
5. Table below gives values of square of integers:

| $x$ | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| $x^2$ | 1 | 4 | 9 | 16 | 25 |

   Using the linear interpolation formula estimate the square of 3.25
   (a) using the points 3 and 4
   (b) using the points 2 and 4
   Compare and comment on the results.
6. Using the data in Exercise 5, estimate the square of 3.25 using the second-order Lagrange formula. Compare the error with the errors obtained in Exercise 5.
7. When the value of $x$ at which we wish to estimate the value of $f(x)$, lies outside the given range, we call it extrapolation. Use the Lagrange formula to find the quadratic equation that takes the following values:

| $x$ | 1 | 2 | 3 |
|---|---|---|---|
| $f(x)$ | 1 | 1 | 2 |

   Find $f(x)$ at $x = 0$ and $x = 4$
8. Given the points below, obtain a cubic polynomial using the Lagrange formula:

| $x$ | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| $f(x)$ | 1 | –1 | –1 | 0 |

9. Find the Lagrange interpolation polynomial which agrees with the following data:

| $x$ | 1.0 | 1.1 | 1.2 |
|---|---|---|---|
| cos $x$ | 0.5403 | 0.4536 | 0.3624 |

   Use it to estimate cos 1.15

10. Find the polynomial of degree three to fit the following points:

| $x$ | $-1$ | $0$ | $1$ | $3$ |
|---|---|---|---|---|
| $f(x)$ | $-6$ | $-2$ | $2$ | $10$ |

11. Show that when $n = 2$, Lagrangian interpolation formula reduces to the linear interpolation formula.
12. The Lagrange interpolation polynomial can be derived directly from Newton's interpolating polynomial. Prove this using the linear case.
13. Fit a second-order Newton's interpolating polynomial to estimate cos 1.15 using the data from Exercise 9.
14. Fit a third-order Newton's interpolating polynomial to estimate cos 1.15 using the data from Exercise 9 along with the additional point cos 1.3 = 0.2675.
15. Given the data

| $x$ | 1.2 | 1.3 | 1.4 | 1.5 |
|---|---|---|---|---|
| $f(x)$ | 1.063 | 1.091 | 1.119 | 1.145 |

(a) Calculate $f(1.35)$ using Newton's interpolating polynomial of order 1 through 3. Choose base points to attain good accuracy.

(b) Comment on the accuracy of results on the order of polynomial.

16. Find the divided differences $f[x_0, x_1]$, $f[x_1, x_2]$ and $f[x_0, x_1, x_2]$ for the data given below.

| $i$ | 0 | 1 | 2 |
|---|---|---|---|
| $x_i$ | 1.0 | 1.5 | 2.5 |
| $f(x_i)$ | 3.2 | 3.5 | 4.5 |

Also find the divided differences $f[x_0, x_2]$ and $f[x_0, x_2, x_1]$. Compare the results $f[x_0, x_1, x_2]$ and $f[x_0, x_2, x_1]$.

17. Estimate the value of ln (3.5) using Newton-Gregory forward difference formula given the following data:

| $x$ | 1.0 | 2.0 | 3.0 | 4.0 |
|---|---|---|---|---|
| ln | 0.0 | 0.6931 | 1.0986 | 1.3863 |

18. Repeat Exercise 17 using Newton's backward difference formula. Compare the accuracy of results.
19. Construct difference tables for the following data:

| $x$ | 0.1 | 0.3 | 0.5 | 0.7 | 0.9 | 1.1 | 1.3 |
|---|---|---|---|---|---|---|---|
| $f(x)$ | 0.003 | 0.067 | 0.148 | 0.248 | 0.370 | 0.518 | 0.697 |

Find $f(0.6)$ using a cube that fits at $x = 0.3$, 0.5, 0.7 and 0.9.

20. What is the minimum degree of polynomial that will exactly fit all seven pairs of data in Exercise 19.
21. Construct a divided difference table for the data in Exercise 19. How do the values compare with those in the table obtained in Exercise 19.
22. State whether the following functions are splines or not.

$$\text{(a) } f(x) = \begin{cases} x & 0 \le x \le 1 \\ \dfrac{x^2+1}{2} & 1 \le x \le 3 \\ 5x-8 & 3 \le x \le 4 \end{cases}$$

$$\text{(b) } f(x) = \begin{cases} x^2-3x+1 & 0 \le x \le 1 \\ x^3+x^2-3 & 1 \le x \le 2 \\ x^3+5x-9 & 2 \le x \le 3 \end{cases}$$

$$\text{(c) } f(x) = \begin{cases} -x+5.5 & 3.0 \le x \le 4.5 \\ 0.64x^2-6.76x+18.46 & 4.5 \le x \le 7.0 \\ -1.6x^2+24.6x-91.3 & 7.0 \le x \le 9.0 \end{cases}$$

23. Find the values of $a$ and $b$ such that the function

$$f(x) = \begin{cases} ax^2-x+1 & 1 \le x \le 2 \\ 3x-b & 2 \le x \le 3 \end{cases}$$

is a quadratic spline.

24. Fit quadratic splines to the data given below:

| $x$ | 1 | 2 | 3 |
|---|---|---|---|
| $f(x)$ | 1 | 1 | 2 |

Predict $f(2.5)$.

25. Develop cubic splines for the data given below and predict $f(1.5)$

| $x$ | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| $f(x)$ | 1 | –1 | –1 | 0 |

26. Given the data points

| $i$ | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| $x_i$ | 1.0 | 3.0 | 4.0 | 7.0 |
| $f(x_i)$ | 1.5 | 4.5 | 9.0 | 25.5 |

Estimate the function value at $x = 1.5$ using cubic splines.

27. The velocity distribution of a fluid near a flat surface is given below:

| $x$ | 0.1 | 0.3 | 0.5 | 0.7 | 0.9 |
|---|---|---|---|---|---|
| $v$ | 0.72 | 1.81 | 2.73 | 3.47 | 3.98 |

$x$ is the distance from the surface (cm) and $v$ is the velocity (cm/sec). Using a suitable interpolation formula obtain the velocity at $x = 0.2$, 0.4, 0.6 and 0.8.

28. The steady-state heat-flow equation $f(x,y)$ is solved numerically and temperature values obtained at the pivotal points of a grid in the domain of interest are tabulated below. (This type of problems are discussed in Chapter 15).

**Table of $f(x, y)$**

| $y$<br>$x$ | 0.5 | 1.0 | 1.5 | 2.0 |
|---|---|---|---|---|
| 0.5 | 15.0 | 21.0 | 25.0 | 31.0 |
| 1.0 | 20.0 | 20.0 | 20.0 | 20.0 |
| 1.5 | 25.5 | 19.0 | 15.0 | 9.0 |
| 2.0 | 30.0 | 20.0 | 10.0 | 0.0 |

Solution of heat-flow equations by numerical methods gives information only at the nodes and not at the intermediate points. We are interested in the temperature at the point (1.25, 1.25). Estimate this value using the data available in the table.

## PROGRAMMING PROJECTS

1. Write subprograms
   (a) COSPLN to compute the coefficients cubic splines, and
   (b) VSPLN to evaluate the spline function at the specified point.
2. Write an interactive main program that will read the given set of table points and the point of interest, estimate the interpolation at the specified point using the subprograms COSPLN and VSPLN developed in Project 1, and then print the results.
3. Write a program to evaluate forward differences and print a forward difference table for a set of $n$ function values.
4. Following is a table that lists values of cube roots of numbers from 1.0 to 2.0 in steps of 0.1.

| $x$ | 1.0 | 1.1 | 1.2 | 1.3 | 1.4 | 1.5 | 1.6 | 1.7 | 1.8 | 1.9 | 2.0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $\sqrt[3]{x}$ | 1.0 | 1.032 | 1.063 | 1.091 | 1.119 | 1.145 | 1.170 | 1.193 | 1.216 | 1.239 | 1.260 |

Write a program for linear interpolation of this table of data and produce another table of cube roots for numbers 1.25 to 1.75 in steps of 0.05 shown as follows:

| $x$ | cube root of $x$ |
|---|---|
| 1.25 | |
| 1.30 | |
| 1.35 | |
| . | |
| . | |
| . | |
| 1.75 | |

5. Modify the program in Project 4 to produce the following table:

| $x$ | Interpolated cube root of $x$ | True value of $\sqrt[3]{x}$ | Error |
|---|---|---|---|
| 1.25 | | | |
| 1.30 | | | |
| . | | | |
| . | | | |
| . | | | |
| 1.75 | | | |

6. Using a table of cosines, accurate to four digits, write a program to implement the following tasks:
   (a) Read the cosine of 0°, 10°, . . . 90°
   (b) Compute the cosine of angle for any value between 0° and 90° using linear interpolation.
   (c) Compare the results of (b) with the output of intrinsic cos function.
7. Write a program to estimate a value $f(x, y)$ from a given table of values of $x$ and $y$ by interpolation.
   Test your program by solving the problem in Exercise 28.

CHAPTER 10

# Curve Fitting: Regression

## 10.1 INTRODUCTION

In the previous chapter we discussed various methods of curve fitting for data points of well-defined functions. In this chapter, we will discuss methods of curve fitting for experimental data.

In many applications, it often becomes necessary to establish a mathematical relationship between experimental values. This relationship may be used for either testing existing mathematical models or establishing new ones. The mathematical equation can also be used to predict or forecast values of the dependent variable. For example, we would like to know the maintenance cost of an equipment (or a vehicle) as a function of age (or mileage) or the relationship between the literacy level and population growth. The process of establishing such relationships in the form of a mathematical equation is known as *regression analysis* or *curve fitting*.

Suppose the values of $y$ for the different values of $x$ are given. If we want to know the effect of $x$ on $y$, then we may write a functional relationship

$$y = f(x)$$

The variable $y$ is called the *dependent* variable and $x$ the *independent variable*. The relationship may be either linear or nonlinear as shown in Fig. 10.1. The type of relationship to be used should be decided by the ›periment based on the nature of scatteredness of data.

It is a standard practice to prepare a *scatter diagram* as shown in Fig. 10.2 and try to determine the functional relationship needed to fit the points. The line should best fit the plotted points. This means that the average error introduced by the assumed line should be minimum. The parameters $a$ and $b$ of the various equations shown in Fig. 10.1 should be evaluated such that the equations best represent the data.

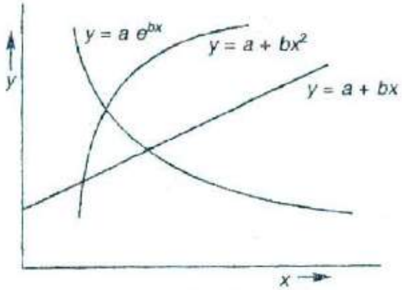


**Fig. 10.1** Various relationships between $x$ and $y$

We shall discuss in this chapter a technique known as *least-squares regression* to fit the data under the following situations:

1. Relationship is linear
2. Relationship is transcendental
3. Relationship is polynomial
4. Relationship involves two or more independent variables

## 10.2 FITTING LINEAR EQUATIONS

Fitting a straight line is the simplest approach of regression analysis. Let us consider the mathematical equation for a straight line

$$y = a + bx = f(x)$$

to describe the data. We know that $a$ is the intercept of the line and $b$ its slope. Consider a point $(x_i, y_i)$ as shown in Fig. 10.2. The vertical distance of this point from the line $f(x) = a + bx$ is the error $q_i$. Then,

$$\begin{aligned} q_i &= y_i - f(x_i) \\ &= y_i - a - bx_i \end{aligned} \tag{10.1}$$

There are various approaches that could be tried for fitting a "best" line through the data. They include:

1. Minimise the sum of errors, i.e., minimise

$$\sum q_i = \sum (y_i - a - bx_i) \tag{10.2}$$

2. Minimise the sum of absolute values of errors

$$\sum |q_i| = \sum |(y_i - a - bx_i)| \tag{10.3}$$

3. Minimise the sum of squares of errors

$$\sum q_i^2 = \sum (y_i - a - bx_i)^2 \tag{10.4}$$

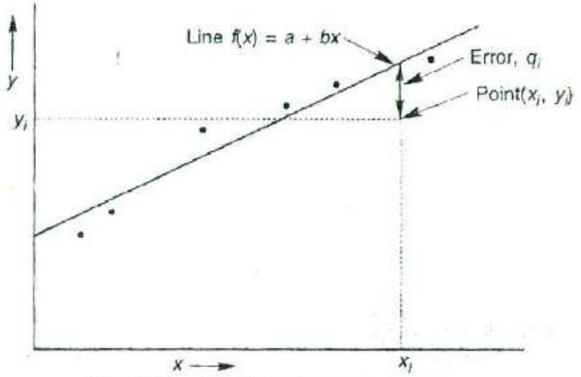


Fig. 10.2 Scatter diagram

It can be easily verified that the first two strategies do not yield a unique line for a given set of data. The third strategy overcomes this problem and guarantees a unique line. The technique of minimising the sum of squares of errors is known as *least squares regression*. In this section we consider the least-squares fit of a straight line.

## Least Squares Regression

Let the sum of squares of individual errors be expressed as

$$Q = \sum_{i=1}^{n} q_i^2 = \sum_{i=1}^{n} [y_i - f(x_i)]^2$$

$$= \sum_{i=1}^{n} (y_i - a - bx_i)^2 \tag{10.5}$$

In the method of least squares, we choose $a$ and $b$ such that $Q$ is minimum. Since $Q$ depends on $a$ and $b$, a necessary condition for $Q$ to be minimum is

$$\frac{\partial Q}{\partial a} = 0 \quad \text{and} \quad \frac{\partial Q}{\partial b} = 0$$

Then

$$\frac{\partial Q}{\partial a} = -2 \sum_{i=1}^{n} (y_i - a - bx_i) = 0$$

$$\frac{\partial Q}{\partial b} = -2 \sum_{i=1}^{n} x_i (y_i - a - bx_i) = 0 \tag{10.6}$$

Thus

$$\sum y_i = na + b \sum x_i$$

$$\sum x_i y_i = a \sum x_i + b \sum x_i^2 \tag{10.7}$$

These are called *normal equations*. Solving for $a$ and $b$, we get

$$\boxed{\begin{aligned} b &= \frac{n\sum x_i y_i - \sum x_i \sum y_i}{n\sum x_i^2 - (\sum x_i)^2} \\ a &= \frac{\sum y_i}{n} - b\frac{\sum x_i}{n} = \bar{y} - b\bar{x} \end{aligned}} \tag{10.8}$$

where $\bar{x}$ and $\bar{y}$ are the averages of $x$ values and $y$ values, respectively.

## Example 10.1

Fit a straight line to the following set of data

| $x$ | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| $y$ | 3 | 4 | 5 | 6 | 8 |

The various summations are given as follows:

| | $x_i$ | $y_i$ | $x_i^2$ | $x_i y_i$ |
|---|---|---|---|---|
| | 1 | 3 | 1 | 3 |
| | 2 | 4 | 4 | 8 |
| | 3 | 5 | 9 | 15 |
| | 4 | 6 | 16 | 24 |
| | 5 | 8 | 25 | 40 |
| $\sum$ | 15 | 26 | 55 | 90 |

Using Eq. (10.8),

$$b = \frac{5 \times 90 - 15 \times 26}{5 \times 55 - 15^2} = 1.20$$

$$a = \frac{26}{5} - 1.20 \times \frac{15}{5} = 1.60$$

Therefore, the linear equation is

$$y = 1.6 + 1.2x$$

The *regression line* along with the data is shown in Fig. 10.3.

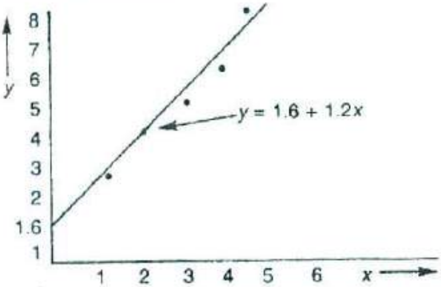


**Fig. 10.3** Plot of the data and regression line of example 10.1

## Algorithm

It is relatively simple to implement the linear regression on a computer. The coefficients *a* and *b* can be evaluated using Algorithm 10.1

**Linear Regression**

```
1. Read data values
2. Compute sum of powers and products
       $\Sigma x_i, \Sigma y_i, \Sigma x_i^2, \Sigma x_i y_i$
3. Check whether the denominator of the equation for $b$ is zero.
4. Compute $b$ and $a$.
5. Print out the equation.
6. Interpolate data, if required.
```

**Algorithm 10.1**

## Program LINREG

Program LINREG implements Algorithm 10.1. The program reads a table of data points and decides a straight line equation to fit the data using the method of least squares regression.

```
* --------------------------------------------------------- *
    PROGRAM LINREG
* --------------------------------------------------------- *
* Main program                                              *
*   This program fits a line Y = A + BX to a given          *
*   set of data points by the method of least squares       *
* --------------------------------------------------------- *
* Functions invoked                                         *
*   ABS                                                     *
* --------------------------------------------------------- *
* Subroutines used                                          *
*   NIL                                                     *
* --------------------------------------------------------- *
* Variables used                                            *
*   X, Y - Data arrays                                      *
*   N - Number of data sets                                 *
*   SUMX - Sum of x values                                  *
*   SUMY - Sum of y values                                  *
*   SUMXX - Sum of squares of x values                      *
*   SUMXY - Sum of products of x and y                      *
*   XMEAN - Mean of x values                                *
*   YMEAN - Mean of y values                                *
*   A - y intercept of the line                             *
*   B - Slope of the line                                   *
* --------------------------------------------------------- *
* Constants used                                          *
*   MAX - Limit for number of data points                 *
* ------------------------------------------------------- *
     INTEGER MAX,N
     REAL X,Y,SUMX,SUMY,SUMXX,SUMXY,XMEAN,YMEAN,DENOM,A,B
     INTRINSIC ABS
     PARAMETER( MAX = 10 )
     DIMENSION X(MAX),Y(MAX)
     WRITE(*,*)
     WRITE(*,*)      'LINEAR REGRESSION'
     WRITE(*,*)
* Reading data values
     WRITE(*,*) 'Input number of data points N'
     READ(*,*) N
     WRITE(*,*)   'Input X and Y values,',
  +               'one set on each line'
     DO 10 I = 1, N
       READ(*,*) X(I), Y(I)
10   CONTINUE
* Computing constants A and B
     SUMX = 0.0
     SUMY = 0.0
     SUMXY = 0.0
     SUMXX = 0.0
     DO 20 I = 1, N
       SUMX = SUMX + X(I)
       SUMY = SUMY + Y(I)
       SUMXX = SUMXX + X(I) * X(I)
       SUMXY = SUMXY + X(I) * Y(I)
20   CONTINUE
     XMEAN = SUMX/N
     YMEAN = SUMY/N
     DENOM = N * SUMXX - SUMX * SUMX
     IF (ABS (DENOM) .GT. 0.00001) THEN
       B = (N * SUMXY - SUMX * SUMY)/DENOM
       A = YMEAN - B * XMEAN
     ELSE
       WRITE(*,*)
       WRITE(*,*) 'NO SOLUTION'
       STOP
     ENDIF
* Printing results
     WRITE(*,*)
```

```
      WRITE(*,*)  'LINEAR REGRESSION LINE Y = A + BX'
      WRITE(*,*)          ,
      WRITE(*,*)  'THE COEFFICIENTS ARE:'
      WRITE(*,*)  '     A = ', A
      WRITE(*,*)  '     B = ', B
      WRITE(*,*)
      STOP
      END
* ---------------- End of main LINREG ---------------- *
```

***Test Run Results*** Shown below is the interactive data input and the linear regression line parameters computed by the program LINREG.

```
               LINEAR REGRESSION
Input number of data points N
5
Input X and Y values, one set on each line
1 3
2 5
3 7
4 9
5 11

LINEAR REGRESSION LINE Y = A + BX

THE COEFFICIENTS ARE:
  A = 1.0000000
  B = 2.0000000
Stop - Program terminated.
```

## 10.3 FITTING TRANSCENDENTAL EQUATIONS

The relationship between the dependent and independent variables is not always linear. Look at Fig. 10.4. The nonlinear relationship between

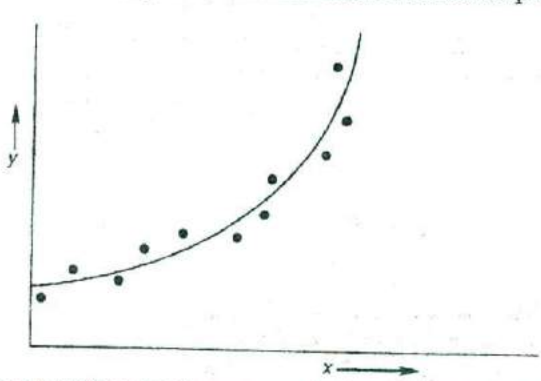


**Fig.10.4** Data would fit a nonlinear curve better than a linear one

them may exist in the form of transcendental equations (or higher order polynomials). For example, the familiar equation for population growth is given by

$$P = p_0 \, e^{kt} \tag{10.9}$$

where $p_0$ is the initial population, $k$ is the rate of growth and $t$ is time. Another example of nonlinear model is the gas low relating to the pressure and volume, as given by

$$p = a \, v^b \tag{10.10}$$

Let us consider Eq. (10.10) first. If we observe values of $p$ for various values of $v$, we can then determine the parameters $a$ and $b$. Using the method of least squares, the sum of the squares of all errors can be written as

$$Q = \sum_{i=1}^{n} [p_i - av_i^b]^2$$

To minimise $Q$, we have

$$\frac{\partial Q}{\partial a} = 0 \qquad \text{and} \qquad \frac{\partial Q}{\partial b} = 0$$

We can prove that

$$\sum p_i v_i^b = a \sum (v_i^b)^2$$

$$\sum p_i v_i^b \ln v_i = a \sum (v_i^b)^2 \ln v_i$$

These equations can be solved for $a$ and $b$. But since $b$ appears under the summation sign, an iterative technique must be employed to solve for $a$ and $b$.

However, this problem can be solved by using the algorithm given in the previous section in the following way: let us rewrite the equation using the conventional variables $x$ and $y$ as

$$y = a x^b$$

If we take logarithm on both the sides, we get

$$\ln y = \ln a + b \ln x \tag{10.11}$$

This equation is similar in form to the linear equation and, therefore, using the same procedure we can evaluate the parameters $a$ and $b$.

$$b = \frac{n \sum \ln x_i \ln y_i - \sum \ln x_i \sum \ln y_i}{n \sum (\ln x_i)^2 - (\sum \ln x_i)^2} \tag{10.12}$$

$$\ln a = R = \frac{1}{n} (\sum \ln y_i - b \sum \ln x_i)$$

$$a = e^R \tag{10.13}$$

Similarly, we can linearise the exponential model shown in Eq. (10.9) by taking logarithm on both the sides. This would yield

$$\ln P = \ln P_0 + kt \ln e$$

Since, $\ln e = 1,$

we have $\ln P = \ln P_o + kt$ (10.14)

This is similar to the linear equation

$$y = a + bx$$

where $y = \ln P$, $a = \ln P_0$, $b = k$, and $x = t$. We can now easily determine $a$ and $b$ and then $P_0$ and $k$.

There is a third form of nonlinear model known as *saturation-growth-rate* equation, as shown below:

$$p = \frac{k_1 t}{k_2 + t} \tag{10.15}$$

This can be linearised by taking inversion of the terms. That is

$$\frac{1}{p} = \left(\frac{k_2}{k_1}\right)\frac{1}{t} + \frac{1}{k_1} \tag{10.16}$$

This is again similar to the linear equation

$$y = a + bx$$

where

$$y = \frac{1}{p}, \qquad x = \frac{1}{t}$$

$$a = \frac{1}{k_1}, \qquad b = \frac{k_2}{k_1}$$

Once we obtain $a$ and $b$, they could be transformed back into the original form for the purpose of analysis.

### Example 10.2

Given the data table

| $x$ | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| $y$ | 0.5 | 2 | 4.5 | 8 | 12.5 |

fit a power-function model of the form

$$y = ax^b$$

Various quantities required in equation (10.12) are tabulated below:

| $x_i$ | $y_i$ | $\ln x_i$ | $\ln y_i$ | $(\ln x_i)^2$ | $(\ln x_i)(\ln y_i)$ |
|---|---|---|---|---|---|
| 1 | 0.5 | 0 | $-0.6931$ | 0 | 0 |
| 2 | 2 | 0.6931 | 0.6931 | 0.4805 | 0.4804 |
| 3 | 4.5 | 1.0986 | 1.5041 | 1.2069 | 1.6524 |
| 4 | 8 | 1.3863 | 2.0794 | 1.9218 | 2.8827 |
| 5 | 12.5 | 1.6094 | 2.5257 | 2.5903 | 4.0649 |
| Sum | | 4.7874 | 6.1092 | 6.1995 | 9.0804 |

Using Eq. (10.12),

$$b = \frac{(5)(9.0804) - (4.7874)(6.1092)}{(5)(6.1995) - (4.7874)^2}$$

$$= \frac{45.402 - 29.2472}{30.9975 - 22.9192}$$

$$= 1.9998$$

$$\ln a = \frac{6.1092 - (1.9998)(4.7847)}{5}$$

$$= -0.6929$$

$$a = 0.5001$$

$\ln a = x$

$e^x = a$

Thus, we obtain the power-function equation as

$$y = 0.5001\ x^{1.9998}$$

Note that the data have been derived from the equation

$$y = \frac{x^2}{2}$$

The discrepancy in the computed coefficients is due to roundoff errors.

## Example 10.3

The temperature of a metal strip was measured at various time intervals during heating and the values are given in the table below:

| time, $t$ (min) | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| temp, $T$ (°$C$) | 70 | 83 | 100 | 124 |

If the relationship between the temperature $T$ and time $t$ is of the form

$$T \quad = be^{t/4} + a$$

estimate the temperature at $t = 6$ min.

We can write the temperature equation in the form

$$y = b\,f(x) + a$$

This is similar to the linear equation except that the variable $x$ is replaced by the function $f(x)$. Therefore, we can solve for the parameters $a$ and $b$ using Eq. (10.8) by replacing

$$x_i \quad \text{by} \quad f(x_i)$$

$$\sum x_i \quad \text{by} \quad \sum f(x_i)$$

$$\sum x_i^2 \quad \text{by} \quad \sum f(x_i)^2$$

Thus,

$$b = \frac{n(\sum f(x_i)y_i) - \sum f(x_i) \sum y_i}{n \sum [f(x_i)]^2 - \left[\sum f(x_i)\right]^2}$$

$$a = \frac{\sum y_i - b \sum f(x_i)}{n}$$

We can set up the following table to obtain the various terms. Note that $f(x) = e^{0.25x}$.

| | $x$ | $y$ | $f(x)$ | $y.\,f(x)$ | $[f(x)]^2$ |
|---|---|---|---|---|---|
| | 1 | 70 | 1.28 | 89.89 | 1.65 |
| | 2 | 83 | 1.65 | 136.84 | 2.72 |
| | 3 | 100 | 2.12 | 211.70 | 4.48 |
| | 4 | 124 | 2.72 | 337.07 | 7.39 |
| Sum | | 377 | 7.77 | 775.5 | 16.24 |

Now,

$$b = \frac{(4)(775.5) - (7.77)(377)}{(4)(16.24) - (7.77)^2}$$

$$= 37.62$$

$$a = \frac{377 - (37.62)(7.77)}{4}$$

$$= 21.16$$

The equation is

$$T = 37.62\, e^{0.25t} + 21.16$$

The temperature, when $t = 6$, is

$$T = 37.62\, e^{0.25(6)} + 21.16$$

$$= 189.76°\text{C}$$

## 10.4 FITTING A POLYNOMIAL FUNCTION

When a given set of data does not appear to satisfy a linear equation, we can try a suitable polynomial as a regression curve to fit the data. The least squares technique can be readily used to fit the data to a polynomial.

Consider a polynomial of degree $m - 1$

$$y = a_1 + a_2 x + a_3 x^2 + \dots + a_m x^{m-1} \qquad (10.17)$$
$$= f(x)$$

If the data contains $n$ sets of $x$ and $y$ values, then the sum of squares of the errors is given by

$$Q = \sum_{i=1}^{n} [y_i - f(x_i)]^2 \tag{10.18}$$

Since $f(x)$ is a polynomial and contains coefficients $a_1$, $a_2$, $a_3$, etc., we have to estimate all the $m$ coefficients. As before, we have the following $m$ equations that can be solved for these coefficients.

$$\frac{\partial Q}{\partial a_1} = 0$$

$$\frac{\partial Q}{\partial a_2} = 0$$

$$\cdots$$

$$\cdots$$

$$\cdots$$

$$\frac{\partial Q}{\partial a_m} = 0$$

Consider a general term,

$$\frac{\partial Q}{\partial a_j} = -2 \sum_{i=1}^{n} [y_i - f(x_i)] \frac{\partial f(x_i)}{\partial a_j} = 0$$

$$\frac{\partial f(x_i)}{\partial a_j} = x_i^{j-1}$$

Thus, we have

$$\sum_{i=1}^{n} [y_i - f(x_i)] x_i^{j-1} = 0 \qquad j = 1, 2, ..., m$$

$$\sum \left[ y_i x_i^{j-1} - x_i^{j-1} f(x_i) \right] = 0$$

Substituting for $f(x_i)$

$$\sum_{i=1}^{n} x_i^{j-1} \left( a_1 + a_2 x_i + a_3 x_i^2 + ... + a_m x_i^{m-1} \right) = \sum_{i=1}^{n} y_i x_i^{j-1}$$

These are $m$ equations ($j = 1, 2...m$) and each summation is for $i = 1$ to $n$.

$$\begin{aligned}
&a_1 n + a_2 \sum x_i + a_3 \sum x_i^2 + ... &&+ a_m \sum x_i^{m-1} = \sum y_i \\
&a_1 \sum x_i + a_2 \sum x_i^2 + a_3 \sum x_i^3 + ... &&+ a_m \sum x_i^m = \sum y_i x_i \\
&\vdots \qquad\qquad \vdots \qquad\qquad \vdots &&\vdots \\
&a_1 \sum x_i^{m-1} + a_2 \sum x_i^m + a_3 \sum x_i^{m+1} + ... &&+ a_m \sum x_i^{2m-2} = \sum y_i x_i^{m-1}
\end{aligned} \tag{10.19}$$

The set of $m$ equations can be represented in matrix notation as follows:

$$\mathbf{CA} = \mathbf{B}$$

where

$$\mathbf{C} = \begin{bmatrix} n & \sum x_i & \sum x_i^2 & \dots & \sum x_i^{m-1} \\ \sum x_i & \sum x_i^2 & \sum x_i^3 & \dots & \sum x_i^m \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ \sum x_i^{m-1} & \sum x_i^m & & \dots & \dots & \sum x_i^{2m-2} \end{bmatrix}$$

$$\mathbf{A} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_m \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} \sum y_i \\ \sum y_i x_i \\ \sum y_i x_2 \\ \vdots \\ \sum y_i x_i^{m-1} \end{bmatrix}$$

The element of matrix **C** is

$$\mathbf{C}(j, k) = \sum_{i=1}^{n} x_i^{j+k-2} \qquad j = 1, 2, \dots, m \quad \text{and} \quad k = 1, 2, \dots, m$$

Similarly,

$$\mathbf{B}(j) = \sum_{i=1}^{n} y_i x_i^{j-1} \qquad j = 1, 2, \dots, m$$

### Example [illegible]

Fit a second order polynomial to the data in the table below:

| $x$ | 1.0 | 2.0 | 3.0 | 4.0 |
|---|---|---|---|---|
| $y$ | 6.0 | 11.0 | 18.0 | 27.0 |

The order of polynomial is 2 and therefore we will have 3 simultaneous equations as shown below:

$$a_1 n + a_2 \sum x_i + a_3 \sum x_i^2 = \sum y_i$$

$$a_1 \sum x_i + a_2 \sum x_i^2 + a_3 \sum x_i^3 = \sum y_i x_i$$

$$a_1 \sum x_i^2 + a_2 \sum x_i^3 + a_3 \sum x_i^4 = \sum y_i x_i^2$$

The sums of powers and products can be evaluated in a tabular form as shown below:

| | $x$ | $y$ | $x^2$ | $x^3$ | $x^4$ | $yx$ | $yx^2$ |
|---|---|---|---|---|---|---|---|
| | 1 | 6 | 1 | 1 | 1 | 6 | 6 |
| | 2 | 11 | 4 | 8 | 16 | 22 | 44 |
| | 3 | 18 | 9 | 27 | 81 | 54 | 162 |
| | 4 | 27 | 16 | 64 | 256 | 108 | 432 |
| $\Sigma$ | 10 | 62 | 30 | 100 | 354 | 190 | 644 |

Substituting these values, we get

$$\begin{aligned} 4a_1 + 10a_2 + 30a_3 &= 62 \\ 10a_1 + 30a_2 + 100a_3 &= 190 \\ 30a_1 + 100a_2 + 354a_3 &= 644 \end{aligned}$$

Solving these equations gives

$$\begin{aligned} a_1 &= 3 \\ a_2 &= 2 \\ a_3 &= 1 \end{aligned}$$

Therefore, the least squares quadratic polynomial is

$$y = 3 + 2x + x^2 \qquad \text{(verify using table data)}$$

## Algorithm for Polynomial Fit

The set of $m$ equations given by Eq. (10.19) can be solved by using an elimination method discussed in Chapter 7. Algorithm 10.2 lists the steps involved in computing the coefficients of the regression polynomial.

### Polynomial Regression

```
1. Read number of data points n and order of polynomial mp
2. Read data values
3. If n ≤ mp,
      print out 'regression is not possible' and stop;
   else
      continue
4. Set m = mp + 1
5. Compute coefficients of C matrix
6. Compute coefficients of B matrix
7. Solve for the coefficients a_1, a_2, ... a_m
8. Write the coefficients
9. Estimate the function value at the given value of independent variable
10. Stop
```

**Algorithm 10.2**

## Program POLREG

This program fits a polynomial curve to a given set of data points by the method of least squares. POLREG uses a subprogram NORMAL to compute the coefficients of normal equations and another subprogram GAUSS to solve the normal equations obtained. Finally, the program prints the polynomial coefficients, $a(1)$ to $a(m)$.

```
* ------------------------------------------------------- *
      PROGRAM POLREG
* ------------------------------------------------------- *
* Main program                                             *
*     This program fits a polynomial curve to a given     *
*     set of data points by the method of least squares   *
* ------------------------------------------------------- *
* Functions invoked                                        *
*     NIL                                                  *
* ------------------------------------------------------- *
* Subroutines used                                         *
*     NORMAL, GAUSS                                        *
* ------------------------------------------------------- *
* Variables used                                           *
*     X,Y - Arrays of data values                          *
*     N - Number of data points                            *
*     MP - Order of the polynomial under construction     *
*     M - Number of polynomial coefficients               *
*     C - Coefficient matrix of normal equations          *
*     B - Right side vector of normal equations           *
*     A - Array of coefficients of the polynomial         *
* ------------------------------------------------------- *
* Constants used                                           *
*     MAX - Maximum number of data points                  *
* ------------------------------------------------------- *
      REAL X,Y,C,A,B
      INTEGER N,MP,M,MAX
      PARAMETER (MAX = 10)
      DIMENSION X(MAX),Y(MAX),C(MAX,MAX),A(MAX),B(MAX)
      WRITE(*,*)
      WRITE(*,*)          'POLYNOMIAL REGRESSION'
      WRITE(*,*)
* Reading data values
      WRITE(*,*) 'Input number of data points(N)'
      READ(*,*) N
      WRITE(*,*) 'Input order of polynomial(MP) required'
      READ(*,*) MP
      WRITE(*,*)  'Input data values X and Y,',
     +            'one set on each line'
```

```
      DO 10 I = 1, N
        READ(*,*) X(I), Y(I)
10    CONTINUE
* Testing the order
      IF(N.LE.MP) THEN
        WRITE(*,*) 'REGRESSION IS NOT POSSIBLE'
        GO TO 20
      ENDIF
* Number of polynomial coefficients
      M = MP+1
* Computation of elements of C and B
      CALL NORMAL(X,Y,C,B,N,M,MAX)
* Computation of coefficients a(1) to a(m)
      CALL GAUSS(M,C,B,A)
* Output of coefficients a(1) to a(m)
      WRITE(*,*)
      WRITE(*,*) 'POLYNOMIAL COEFFICIENTS'
      WRITE(*,*)
      WRITE(*,*) (A(I), I=1,M)
      WRITE(*,*)
20    STOP
      END
* ------------ End of main program POLREG -----------  *
* ------------------------------------------------------ *
      SUBROUTINE NORMAL(X,Y,C,B,N,M,MAX)
* ------------------------------------------------------ *
* Subroutine                                             *
*   This subroutine computes the coefficients            *
*   of normal equations                                  *
* ------------------------------------------------------ *
* Arguments                                              *
* Input                                                  *
*   N - Number of data points                            *
*   X,Y - Arrays of data values                          *
*   M - Number of coefficients of the polynomial         *
*   MAX - Maximum size of arrays                         *
* Output                                                 *
*   C - Coefficient matrix of normal equations           *
*   B - Right side vector of normal equations            *
* ------------------------------------------------------ *
* Local Variables                                        *
*   NIL                                                  *
* ------------------------------------------------------ *
* Functions invoked                                         *
*   NIL                                                     *
* ----------------------------------------------------------*
* Subroutines called                                        *
*   NIL                                                     *
* ----------------------------------------------------------*

      REAL   X,Y,C,B
      INTEGER   N,M,MAX
      DIMENSION X(MAX),Y(MAX),C(MAX,MAX),B(MAX)

      DO 30 J=1,M
        DO 20 K=1,M
           C(J,K) = 0.0
           L1 = K+J-2
           DO 10 I=1,N
             C(J,K) = C(J,K) + X(I) ** L1
10         CONTINUE
20      CONTINUE
30    CONTINUE

      DO 50 J= 1,M
          B(J) = 0.0
          L2 = J-1
          DO 40 I = 1,N
             B(J) = B(J) + Y(I) * X(I) ** L2
40        CONTINUE
50    CONTINUE

      RETURN
      END
* ------------- End of subroutine NORMAL ------------       *
* ----------------------------------------------------------*

      SUBROUTINE GAUSS(N,A,B,X)
* ----------------------------------------------------------*
* Subroutine                                                *
*   This subroutine solves a set of n linear                *
*   equations by Gauss elimination method                   *
* ----------------------------------------------------------*
* Arguments                                                 *
* Input                                                     *
*   N - Number of equations                                 *
*   A - Matrix of coefficients                              *
*   B - Right side vector                                   *
* Output                                                    *
*   X - Solution vector                                     *
* ----------------------------------------------------------*
* Local Variables                                           *
*   PIVOT, FACTOR, SUM                                      *
* ----------------------------------------------------------*
* Functions invoked                                         *
*   NIL                                                     *
* --------------------------------------------------------- *
* Subroutines called                                        *
*   NIL                                                     *
* --------------------------------------------------------- *
    INTEGER N
    REAL A,B,X, PIVOT,FACTOR,SUM
    DIMENSION A(10,10), B(10), X(10)
* --------------- Elimination begins --------------- *
    DO 33 K = 1, N-1
       PIVOT = A(K,K)
       DO 22 I = K+1, N
          FACTOR = A(I,K)/PIVOT
          DO 11 J = K+1, N
             A(I,J) = A(I,J) - FACTOR * A(K,J)
11       CONTINUE
         B(I) = B(I) - FACTOR * B(K)
22     CONTINUE
33  CONTINUE
* ------------- Back substitution begins ------------- *
    X(N) = B(N)/A(N,N)
    DO 55 K = N-1,1,-1
       SUM = 0
       DO 44 J = K+1,N
          SUM = SUM + A(K,J) * X(J)
44     CONTINUE
       X(K) = (B(K) - SUM)/A(K,K)
55  CONTINUE
    RETURN
    END
* -------------- End of subroutine GAUSS ------------- *
```

***Test Run Results*** The program was used to fit a polynomial curve to the following data points:

| $x_i$ | 1.0 | 2.1 | 3.2 | 4.0 |
|---|---|---|---|---|
| $y_i$ | 2.0 | 2.5 | 3.0 | 4.0 |

The results are given below:

```
               POLYNOMIAL REGRESSION
Input number of data points(N)
4
Input order of polynomial(MP) required
2
Input data values X and Y, one set on each line
1.0 2.0
```

```
2.1 2.5
3.2 3.0
4.0 4.0

POLYNOMIAL COEFFICIENTS

        2.0740160    -2.053067E-001   1.680441E-001

Stop - Program terminated.
```

## MULTIPLE LINEAR REGRESSION

There are a number of situations where the dependent variable is a function of two or more variables. For example, the salary of a salesperson may be expressed as

$$y = 500 + 5x_1 + 8x_2$$

where $x_1$ and $x_2$ are the number of units sold of products 1 and 2, respectively. We shall discuss here an approach to fit the experimental data where the variable under consideration is a linear function of two independent variables.

Let us consider a two-variable linear function as follows:

$$y = a_1 + a_2x + a_3z \tag{10.20}$$

The sum of the squares of errors is given by

$$Q = \sum_{i=1}^{n}(y_i - a_1 - a_2x_i - a_3z_i)^2$$

Differentiating with respect to $a_1$, $a_2$ and $a_3$, we get,

$$\frac{\partial Q}{\partial a_1} = -2\sum(y_i - a_1 - a_2x_i - a_3z_i)$$

$$\frac{\partial Q}{\partial a_2} = -2\sum(y_i - a_1 - a_2x_i - a_3z_i)\,x_i$$

$$\frac{\partial Q}{\partial a_3} = -2\sum(y_i - a_1 - a_2x_i - a_3z_i)\,y_i$$

Setting these partial derivatives equal to zero results in

$$na_1 + \left(\sum x_i\right)a_2 + \left(\sum z_i\right)a_3 = \sum y_i$$

$$\left(\sum x_i\right)a_1 + \left(\sum x_i^2\right)a_2 + \left(\sum x_iz_i\right)a_3 = \sum y_ix_i$$

$$\left(\sum z_i\right)a_1 + \left(\sum x_iz_i\right)a_2 + \left(\sum z_i^2\right)a_3 = \sum y_iz_i$$

These are three simultaneous equations with three unknowns and, therefore, can be expressed in matrix form as

$$\begin{bmatrix} n & \sum x_i & \sum z_i \\ \sum x_i & \sum x_i^2 & \sum x_iz_i \\ \sum z_i & \sum x_iz_i & \sum z_i^2 \end{bmatrix}\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} \sum y_i \\ \sum y_ix_i \\ \sum y_iz_i \end{bmatrix} \tag{10.21}$$

This equation can be solved using any standard method. This is a two-dimensional case and, therefore, we obtain a regression "plane" rather than"line".

We can easily extend Eq. (10.21) to the more general case

$$y = a_1 + a_2x_1 + a_3x_2 + \ldots + a_{m+1}\, x_m$$

**Example 10.5**

Given the table of data

| $x$ | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| $z$ | 0 | 1 | 2 | 3 |
| $y$ | 12 | 18 | 24 | 30 |

Obtain a regression plane to fit the data.

The various sums of powers and products required for evaluation of coefficients are tabulated below:

| | $x$ | $z$ | $y$ | $x^2$ | $z^2$ | $xz$ | $yx$ | $yz$ |
|---|---|---|---|---|---|---|---|---|
| | 1 | 0 | 12 | 1 | 0 | 0 | 12 | 0 |
| | 2 | 1 | 18 | 4 | 1 | 2 | 36 | 18 |
| | 3 | 2 | 24 | 9 | 4 | 6 | 72 | 48 |
| | 4 | 3 | 30 | 16 | 9 | 12 | 120 | 90 |
| $\Sigma$ | 10 | 6 | 84 | 30 | 14 | 20 | 240 | 156 |

On substitution of these values in Eq. (10.21) we get

$$\left.\begin{aligned} 4a_1 + 10a_2 + 5a_3 &= 84 \\ 10a_1 + 30a_2 + 20a_3 &= 240 \\ 6a_1 + 20a_2 + 14a_3 &= 156 \end{aligned}\right\}$$

Solution of these equations results in

$$a_1 = 10$$
$$a_2 = 2$$
$$a_3 = 4$$

Thus, the regression plane is

$$y = 10 + 2x + 4z$$

## 10.6 ILL-CONDITIONING IN LEAST SQUARES METHODS

The problem of ill-conditioning can arise in implementing the least squares regression methods. As a consequence, the computed solution might differ substantially from its exact solution. This problem becomes more severe when the degree of approximating polynomial is large.

Ill-conditioning arises basically due to very large differences in the coefficients of the normal equations. Recall that the coefficients are sums of powers and products of data values. Techniques such as pivoting and iterative refinement can be incorporated to overcome the problem of ill-conditioning. The problem of ill-conditioning can also be tackled by increasing the precision of arithmetic operations.

Another way of overcoming the least-squares ill-conditioning problem is to use orthogonal polynomials. This would enable us to obtain the coefficients $a_i$ in closed form, thus avoiding numerical solution of simultaneous equations. Further discussions on this approach is beyond the scope of this book.

## 10.7 SUMMARY

We often use experimental data for establishing a relationship between two variables. This relationship may be used for testing some existing mathematical models or establishing new ones or even estimating the values of dependent variables at some point. In this chapter, we have used a technique known as least squares regression to establish the following types of relationship between the variables of a table of experimental data.

- Linear relationship
- Transcendental relationship
- Polynomial relationship
- Multivariable relationship

Also presented are FORTRAN programs and test results for obtaining linear and polynomial equations for experimental data.

### Key Terms

| | |
|---|---|
| *Curve fitting* | *Regression analysis* |
| *Dependent variable* | *Regression line* |
| *Independent variable* | *Regression plane* |
| *Least squares regression* | *Saturation growth rate* |
| *Normal equations* | *Scatter diagram* |

### REVIEW QUESTIONS

1. What is regression analysis?
2. What is a scatter diagram?
3. What is the principle of least squares regression?
4. Show that the linear regression line of $y$ on $x$ passes through the point that represents the mean of $x$ and $y$ values.

5. Derive normal equations for evaluating the parameters $a$ and $b$ to fit data to

   (a) power-function model of the form

   $$y = ax^b$$

   (b) population growth model of the form

   $$y = a\, e^{bx}$$

   using the principle of least squares.

6. Draw a flow chart to illustrate the steps involved in developing a program for multiple regression.

## REVIEW EXERCISES

1. Use least squares regression to fit a straight line to the data.

| $x$ | 1 | 3 | 4 | 6 | 8 | 9 | 11 |
|---|---|---|---|---|---|---|---|
| $y$ | 1 | 2 | 4 | 4 | 5 | 7 | 8 |

Along with the slope and intercept, also compute the standard error of the estimate.

2. In an organisation, systematic efforts were introduced to reduce the employee absenteeism and results for the first 10 months are shown below:

| Months | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| Absentees (per cent) | 10 | 9 | 9 | 8.5 | 9 | 8 | 8.5 | 7 | 8 | 7.5 |

Fit a linear least squares line to the data and from this equation estimate the average weekly reduction in absenteeism.

3. The following table shows heights ($h$) and weights ($w$) of 8 persons.

| $h$(cm) | 175 | 165 | 160 | 180 | 150 | 170 | 155 | 185 |
|---|---|---|---|---|---|---|---|---|
| $w$(kg) | 68 | 58 | 59 | 71 | 51 | 62 | 53 | 68 |

Assuming a linear relationship between the height and weight, find the regression line and estimate the weights of the persons with the following heights.

(a) 140 cm (b) 163 cm (c) 172.5

4. Fit a geometric curve

$$y = ax^b$$

to the following data:

| $x$ | $-2$ | $-1$ | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|---|---|
| $y$ | 38 | 6 | 0 | $-5$ | $-41$ | 130 | 300 |

5. Given the table of points

| $x$ | 0 | 2 | 4 | 6 | 8 | 12 | 16 | 20 |
|---|---|---|---|---|---|---|---|---|
| $y$ | 10 | 12 | 18 | 22 | 20 | 30 | 26 | 30 |

use least squares regression to fit

(a) straight line, and

(b) parabola

to the data. Compute and compare the errors.

6. Fit the saturation growth rate model

$$y = \frac{ax}{b+x}$$

to the data given below.

| $x$ | 2 | 4 | 6 | 8 |
|---|---|---|---|---|
| $y$ | 1.4 | 2.0 | 2.4 | 2.6 |

7. Fit the power equation

$$y = ax^b$$

to the data given in Exercise 6.

8. Fit a quadratic polynomial to the data given in Exercise 6.

9. Use the power equation to the data

| $x$ | 7.5 | 10 | 12.5 | 15 | 17.5 | 20 |
|---|---|---|---|---|---|---|
| $y$ | 2.4 | 1.6 | 1.2 | 0.8 | 0.6 | 0.6 |

10. Use the exponential model

$$y = a\,e^{bx}$$

to fit the data

| $x$ | 0.4 | 0.8 | 1.2 | 1.6 | 2.0 | 2.4 |
|---|---|---|---|---|---|---|
| $y$ | 75 | 100 | 140 | 200 | 270 | 375 |

11. Fit a parabola to the data given in Exercise 10.

12. Find the least squares line $y = ax + b$ that fits the following data, assuming that there are no errors in $x$ values.

| $x$ | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| $y$ | 4.05 | 7.12 | 9.65 | 12.20 | 15.20 | 19.00 |

13. In Exercise 12, treat $x$ as dependent variable on $y$ and find the least squares line $x = ay + b$, assuming that there are no errors in $y$ but $x$ values contain errors. Observe that this is not the same line obtained in Exercise 12.

14. Use multiple linear regression to fit

| $x_1$ | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| $x_2$ | 4 | 3 | 2 | 1 | 0 |
| $y$ | 18 | 16 | 16 | 12 | 10 |

Compute coefficients and the error of estimate.

15. Given the data points

| | | | | | |
|---|---|---|---|---|---|
| $x_1$ | 5 | 4 | 3 | 2 | 1 |
| $x_2$ | 3 | –2 | –1 | 4 | 0 |
| $y$ | 15 | –8 | –1 | 26 | 8 |

obtain a regression plane to fit the data.

## PROGRAMMING PROJECTS

1. Modify the program LINREG to calculate the sum of squares of the errors for the linear fit and print the error output.
2. A set of data, when plotted resembles an exponential curve

$$y = a(b^x)$$

Write a program to evaluate the parameters $a$ and $b$ of this regression curve using the principle of least squares.

3. In fitting a polynomial, its degree should be chosen such that the error is minimum. Given a set of data, it would be difficult to decide the degree that would represent the data best. A good rule of thumb is to begin with the first degree and continue fitting higher order polynomials until

$$\frac{Q_i}{n-i-1} > \frac{Q_{i-1}}{n-i}$$

or until a polynomial of degree $n$–1 is obtained. $Q_i$ is the sum of squares of errors of polynomial of degree $i$.

   (a) Prepare a flow chart to fit a polynomial that satisfies this condition.

   (b) Modify the program POLREG to incorporate these changes.

4. Develop a user-friendly program for multiple regression.
5. Develop a user-friendly, menu-driven program that allows us an option to select and use one of the following models to fit a given set of data.

   (a) Straight line model

   (b) Exponential model

   (c) Power equation

   (d) Saturation-growth rate model

CHAPTER 11

# Numerical Differentiation

## 11.1 NEED AND SCOPE

Need for differentiation of a function arises quite often in engineering and scientific problems. If the function has a closed form representation in terms of standard calculus, then its derivatives can be found exactly. However, in many situations, we may not know the exact function. What we know is only the values of the function at a discrete set of points. For instance, we are given the distance travelled by a moving object at some regular time intervals and asked to determine its velocity at a particular time. In some other instances, the function is known but it is so complicated that an analytic differentiation is difficult (if not impossible). In both these situations, we seek the help of numerical techniques to obtain the estimates of function derivatives. The method of obtaining the derivative of a function using a numerical technique is known as *numerical differentiation*. There are essentially two situations where numerical differentiation is required. They are :

1. The function values are known but the function is unknown. Such functions are called *tabulated function*.
2. The function to be differentiated is complicated and, therefore, it is difficult to differentiate.

In this chapter, we discuss various numerical differentiation methods that could be applied to both tabulated and continuous functions.

Remember that while analytical methods give exact answers, the numerical techniques provide only approximations to derivatives. Numerical differentiation methods are very sensitive to roundoff errors, in addition to the truncation error introduced by the methods themselves. Therefore, we also discuss the errors and ways to minimise them.

## 11.2 DIFFERENTIATING CONTINUOUS FUNCTIONS

We discuss here the numerical process of approximating the derivative $f'(x)$ of a function $f(x)$, when the function itself is available.

### Forward Difference Quotient

Consider a small increment $\Delta x = h$ in $x$. According to Taylor's theorem, we have

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2} f''(\theta) \tag{11.1}$$

for $x \leq \theta \leq x + h$. By rearranging the terms, we get

$$f'(x) = \frac{f(x+h) - f(x)}{h} - \frac{h}{2} f''(\theta) \tag{11.2}$$

Thus, if $h$ is chosen to be sufficiently small, $f'(x)$ can be approximated by

$$\boxed{f'(x) = \frac{f(x+h) - f(x)}{h}} \tag{11.3}$$

with a truncation error of

$$\boxed{E_t(h) = -\frac{h}{2} f''(\theta)} \tag{11.4}$$

Equation (11.3) is called the first order *forward difference quotient*. This is also known as *two-point formula*. The truncation error is in the order of $h$ and can be decreased by decreasing $h$.

Similarly, we can show that the first-order *backward difference quotient* is

$$\boxed{f'(x) = \frac{f(x) - f(x-h)}{h}} \tag{11.5}$$

### Example 11.1

Estimate approximate derivative of $f(x) = x^2$ at $x = 1$, for $h = 0.2$, 0.1, 0.05 and 0.01 using the first-order forward difference formula.

$$f'(x) = \frac{f(x+h) - f(x)}{h}$$

Therefore,

$$f'(1) = \frac{f(1+h) - f(1)}{h}$$

Derivative approximations are tabulated below:

| $h$ | $f'(1)$ | *Error* |
|---|---|---|
| 0.2 | 2.2 | 0.2 |
| 0.1 | 2.1 | 0.1 |
| 0.05 | 2.05 | 0.05 |
| 0.01 | 2.01 | 0.01 |

Note that the correct answer is 2. The derivative approximation approaches the exact value as $h$ decreases. The truncation error decreases proportionally with decrease in $h$. There is no roundoff error.

## Central Difference Quotient

NUB

Note that Eq. (11.3) was obtained using the linear approximation to $f(x)$. This would give large truncation errors if the functions were of higher order. In such cases, we can reduce truncation errors for a given $h$ by using a quadratic approximation, rather than a linear one. This can be achieved by taking another term in Taylor's expansion, i.e.,

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2!} f''(x) + \frac{h^3}{3!} f'''(\theta_1) \qquad (11.6)$$

Similarly,

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2!} f''(x) - \frac{h^3}{3!} f'''(\theta_1) \qquad (11.7)$$

Subtracting Eq. (11.7) from Eq. (11.6), we obtain

$$f(x+h) - f(x-h) = 2hf'(x) + \frac{h^3}{3!} [f'''(\theta_1) + f'''(\theta_2)] \qquad (11.8)$$

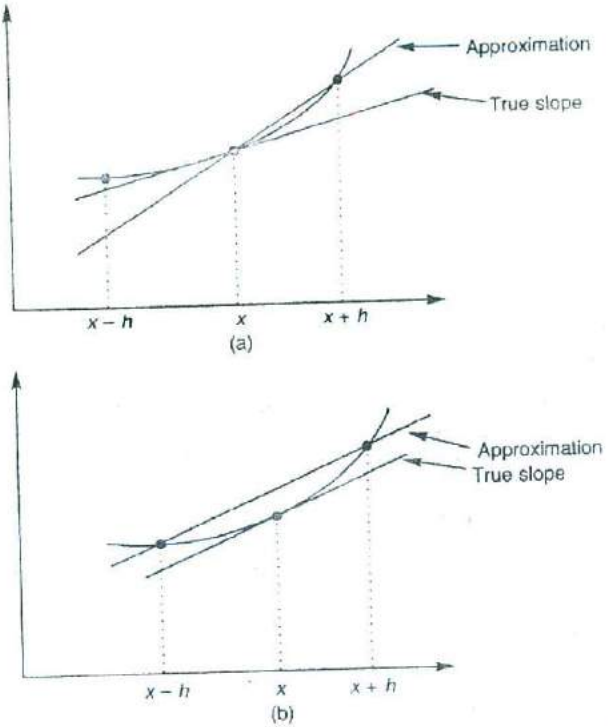
Thus, we have

$$\boxed{f'(x) = \frac{f(x+h) - f(x-h)}{2h}} \qquad (11.9)$$

with the truncation error of

$$E_t(h) = -\frac{h^2}{12} [f'''(\theta_1) + f'''(\theta_2)] = -\frac{h^2}{6} f'''(\theta)$$

which is of order $h^2$. Equation (11.9) is called the second-order *central difference quotient*. Note that this is the average of the forward difference quotient and the backward difference quotient. This is also known as *three-point formula*. The distinction between the two-point and three-point formulae is illustrated in Fig. 11.1(a) and Fig. 11.1(b). Note that the approximation is better in the case of three-point formula.



(a)

(b)

Fig. 11.1 Illustration of (a) Two-point formula and (b) Three-point formula

### Example 11.2

Repeat the exercise given in Example 11.1 for the three-point formula.

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h}$$

Therefore,

$$f'(1) = \frac{f(1+h) - f(1-h)}{2h}$$

The derivative approximations are tabulated below:

| $h$ | $f'(1)$ | *Error* |
|---|---|---|
| 0.2 | 2.0 | 0 |
| 0.1 | 2.0 | 0 |
| 0.05 | 2.0 | 0 |

The derivative is exact for all values of $h$. This is because we have used quadratic approximation for a quadratic function. We can also derive further higher-order derivatives by using more points in the formula. For example, the *five-point central difference formula* is given by

$$\boxed{f'(x) = \frac{-f(x+2h) + 8f(x+h) - 8f(x-h) + f(x-2h)}{12h}} \qquad (11.10)$$

This is a fourth-order approximation and the truncation error is of order $h^4$. In this case, the truncation error will approach zero much faster compared to the three-point approximation. The derivation of Eq. (11.10) is left to the reader as an exercise. (*Hint: use step size 2h instead of h in Eq. (11.8) and use up to fifth derivative of Taylor's expansion*).

## Error Analysis

As mentioned earlier, numerical differentiation is very sensitive to roundoff errors. If $E_r(h)$ is the roundoff error introduced in an approximate derivative, then the total error is given by

$$E(h) = E_r(h) + E_t(h)$$

Let us consider the two-point formula for the purpose of analysis. That is,

$$f'(x) = \frac{f(x+h) - f(x)}{h} = \frac{f_1 - f_0}{h}$$

If we assume the roundoff errors in $f_1$ and $f_0$ as $e_1$ and $e_0$, respectively, then

$$f'(x) = \frac{(f_1 + e_1) - (f_0 + e_0)}{h}$$

$$= \frac{f_1 - f_0}{h} + \frac{e_1 - e_0}{h}$$

If the errors $e_1$ and $e_0$ are of the magnitude $e$ and of opposite sign (i.e., the worst case) then we get the bound for roundoff error as

$$|E_r(h)| \leq \frac{2e}{h}$$

We know that the truncation error for two-point formula is

$$|E_t(h)| = -\frac{h}{2} f''(\theta)$$

or

$$|E_t(h)| \leq \frac{M_2 h}{2}$$

where $M_2$ is the bound given by

$$M_2 = \max_{x \leq \theta \leq x+h} |f''(\theta)|$$

Thus, the bound for total error in the derivative is

$$\boxed{|E(h)| \leq \frac{M_2 h}{2} + \frac{2e}{h}} \tag{11.11}$$

Note that when the step size $h$ is increased, the truncation error increases while the roundoff error decreases. This is illustrated in Fig. 11.2. For small values of $h$, roundoff error has an overriding influence on the total error. Therefore, while reducing the step size, we should exercise proper judgement in choosing the size. This argument applies to all the formulae discussed here.

Fig. 11.2 Error in derivatives as a function of $h$

We can obtain a rough estimate of $h$ that gives the minimum error. By differentiating Eq. (11.11) with respect to $h$, we obtain

$$E'(h) = \frac{M_2}{2} - \frac{2e}{h^2}$$

We know that $E(h)$ is minimum when $E'(h) = 0$. That is,

$$\frac{M_2}{2} - \frac{2e}{h^2} = 0$$

Solving for $h$, we obtain

$$h_{\text{opt}} = 2\sqrt{\frac{e}{M_2}} \tag{11.12}$$

Substituting this in Eq. (11.11), we get

$$E(h_{\text{opt}}) = 2\sqrt{eM_2} \tag{11.13}$$

F

### Example 11.3

**Compute the approximate derivatives of $f(x) = \sin x$, at $x = 0.45$ radians, at increasing values of $h$ from 0.01 to 0.04, with a step size of 0.005. Analyse the total error. What is the optimum step size?**

$$f(x) = \sin x$$

Using two-point formula

$$f'(x) = \frac{f(x+h) - f(x)}{h}$$

.46

Given

$$x = 0.45 \text{ radians}$$

So, $f(x) = \sin(0.45) = 0.4350$ (rounded to four digits). Exact $f'(x) = \cos x = \cos(0.45) = 0.9004$.

Table below gives the approximate derivatives of $\sin x$ at $x = 0.45$ using various values of $h$.

| $h$ | $f(x+h)$ | $f'(x)$ | *Error* |
|---|---|---|---|
| 0.010 | 0.4439 | 0.8900 | 0.0104 |
| 0.015 | 0.4484 | 0.8933 | 0.0071 |
| 0.020 | 0.4529 | 0.8950 | 0.0054 |
| 0.025 | 0.4573 | 0.8935 | 0.0069 |
| 0.030 | 0.4618 | 0.8933 | 0.0071 |
| 0.035 | 0.4662 | 0.8914 | 0.0090 |
| 0.040 | 0.4706 | 0.8900 | 0.0104 |

The table shows that the total error decreases from 0.0104 (at $h = 0.01$) till $h = 0.02$ and again increases when $h$ is increased as illustrated in Fig. 11.2.

Since we have used four significant digits, the bound for roundoff error $e$ is $0.5 \times 10^{-4}$. For the two-point formula, the bound $M_2$ is given by

$$M_2 = \max |f''(\theta)|$$

$$0.45 \le \theta \le 0.49,$$

$$= |\sin(0.49)| = 0.4706$$

Therefore, the optimum step size is

$$h_{\text{opt}} = 2\sqrt{\frac{e}{M_2}} = 2\sqrt{\frac{0.5 \times 10^{-4}}{0.4706}}$$

$$= 0.0206$$

This agrees very closely with our results.

## Higher-order Derivatives

We can also obtain approximations to higher-order derivatives using Taylor's expansion. To illustrate this, we derive here the formula for $f''(x)$. We know that

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2!} f''(x) + \frac{h^3}{3!} f'''(x) + R_1$$

and

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2!} f''(x) - \frac{h^3}{3!} f'''(x) + R_2$$

Adding these two expansions gives

$$f(x+h) + f(x-h) = 2f(x) + h^2 f''(x) + R_1 + R_2$$

Therefore

$$f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} - \frac{(R_1 + R_2)}{h^2}$$

Thus, the approximation to second derivative is

$$\boxed{f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}} \qquad (11.14)$$

The truncation error is

$$\begin{aligned} E_t(h) &= -\frac{R_1 + R_2}{h^2} \\ &= -\frac{1}{h^2} \frac{h^4}{4!} (f^{(4)}(\theta_1) + f^{(4)}(\theta_2)) \\ &= -\frac{h^2}{12} f^{(4)}(\theta) \end{aligned}$$

The error is of order $h^2$.

Similarly, we can obtain other higher-order derivatives with the errors of order $h^3$ and $h^4$.

**Example 11.4** F// NUB

Find approximation to second derivative of cos $(x)$ at $x = 0.75$ with $h = 0.01$. *Compare with the* true value.

$$f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}$$

$$f''(0.75) = \frac{f(0.76) - 2f(0.75) + f(0.74)}{0.0001} \quad (\text{at } h = 0.01)$$

$$= \frac{0.7248360 - 2(0.7316888) + 0.7384685}{0.0001}$$

$$= \frac{1.4633046 - 1.4633776}{0.0001}$$

$$= -0.7300000$$

Exact value of $f''(0.75) = -\cos(0.75)$

$$= -0.7316888$$

$$\text{Error} = -0.0016888$$

This error includes roundoff error as well.

## 11.3 DIFFERENTIATING TABULATED FUNCTIONS

Suppose that we are given a set of data points $(x_i, f_i)$, $i = 0, 1, \dots n$ which correspond to the values of an unknown function $f(x)$ and we wish to estimate the derivatives at these points. Assume that the points are equally spaced with a step size of $h$.

When function values are available in tabulated form, we may approximate this function by an interpolation polynomial $p(x)$ discussed in Chapter 9 and then differentiate $p(x)$. We will use here Newton's divided difference interpolation polynomial.

Let us first consider the linear equation

$$p_1(x) = a_0 + a_1(x - x_0) + R_1$$

where $R_1$ is the remainder term used for estimation. Upon differentiation of this formula, we obtain

$$p_1'(x) = a_1 + \frac{\mathrm{d}R_1}{\mathrm{d}x}$$

Then the approximate derivative of the function $f(x)$ is given by

$$f'(x) = p_1'(x) = a_1$$

We know that

$$a_1 = f[x_0, x_1]$$

$$= \frac{f(x_1) - f(x_0)}{x_1 - x_0}$$

On substituting

$$h = x_1 - x_0$$

$$x_1 = x + h$$

$$x_0 = x$$

we get

$$\boxed{f'(x) = \frac{f(x+h) - f(x)}{h}} \qquad (11.15)$$

This is the familiar two-point forward difference formula.

Now, let us consider the quadratic approximation. Here, we need to use three points. Thus,

$$p_2(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + R_2$$

Then

$$p_2'(x) = a_1 + a_2[(x - x_0) + (x - x_1)] + \frac{dR_2}{dx}$$

Thus, we obtain

$$f'(x) = a_1 + a_2[(x - x_0) + (x - x_1)] \tag{11.16}$$

Let $x_0 = x$, $x_1 = x + h$, $x_2 = x + 2h$, Then

$$a_1 = \frac{f(x+h) - f(x)}{h}$$

$$a_2 = f[x_0, x_1, x_2] = \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0}$$

$$= \frac{\dfrac{f(x_2) - f(x_1)}{x_2 - x_1} - \dfrac{f(x_1) - f(x_0)}{x_1 - x_0}}{x_2 - x_0}$$

$$= \frac{f(x+2h) - 2f(x+h) + f(x)}{2h^2}$$

Substituting for $a_1$ and $a_2$ in Eq. (11.16) and after simplification, we get

$$\boxed{f'(x) = \frac{-3f(x) + 4f(x+h) - f(x+2h)}{2h}} \tag{11.17}$$

This is a three-point forward difference formula. We can obtain a three-point backward difference formula by replacing $h$ by $-h$ in Eq. (11.17). Therefore, the three-point backward difference formula is given by

$$\boxed{f'(x) = \frac{3f(x) - 4f(x-h) - f(x-2h)}{2h}} \tag{11.18}$$

Similarly, we can obtain the three-point central difference formula by letting $x_0 = x$, $x_1 = x - h$, $x_2 = x + h$ in Eq. (11.16). Thus,

$$a_1 = \frac{f(x) - f(x-h)}{h}$$

$$a_2 = \frac{f(x+h) - 2f(x) + f(x-h)}{2h^2}$$

Substituting these values in Eq. (11.16) we get

$$\boxed{f'(x) = \frac{f(x+h) - f(x-h)}{2h}} \tag{11.19}$$

## Error Analysis

Let us first take the linear case.

$$R_1 = f[x_0, x_1, x]\,(x - x_0)\,(x - x_1)$$

where

$$f[x_0, x_1, x] = \frac{f''(\theta)}{2!}$$

for some point $q$ in the interval containing $x_0$, $x_1$ and $x$. Then

$$\frac{dR_1}{dx} = \frac{f''(\theta)}{2}\,[(x - x_0) + (x - x_1)]$$

Letting $x_0 = x$ and $x_1 = x + h$

$$\frac{dR_1}{dx} = -\frac{h}{2} f''(\theta), \qquad x \le \theta \le x + h$$

Therefore, the truncation error is of order $h$. This conforms with Eq. (11.4). Now, let us consider the quadratic approximation.

$$R_2 = f[x_0, x_1, x_2, x]\,(x - x_0)\,(x - x_1)\,(x - x_2)$$

$$f[x_0, x_1, x_2, x] = \frac{f'''(\theta)}{3!}$$

for some point $q$ in the interval containing $x_0$, $x_1$, $x_2$ and $x$.

$$\frac{dR_2}{dx} = \frac{f'''(\theta)}{3!}\,[(x - x_0)\,(x - x_1) + (x - x_1)\,(x - x_2) + (x - x_2)\,(x - x_0)]$$

By setting $x_0 = x$, $x_1 = x + h$ and $x_2 = x + 2h$,

$$\frac{dR_2}{dx} = \frac{h^2}{3} f'''(\theta), \qquad x \le \theta \le x + h$$

This error equation holds good for both forward and backward three-point formulae.

For central difference formula, we must set $x_0 = x$, $x_1 = x - h$ and $x_2 = x + h$. Therefore,

$$\frac{dR_2}{dx} = -\frac{h^2}{6} f'''(\theta), \qquad x - h \le \theta \le x + h$$

Note that the error is of order $h^2$

### Example 11.5

E, ✓✓

The table below gives the values of distance travelled by a car at various time intervals during the initial running

| Time, $t$(s) | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|
| Distance travelled, $s(t)$ (km) | 10.0 | 14.5 | 19.5 | 25.5 | 32.0 |

Estimate velocity at time $t = 5$, $t = 7$ and $t = 9$.

We know that velocity is given by the first derivative of $s(t)$. At $t = 5$, we use the three-point forward difference formula (11.17).

$$v(t) = \frac{-3s(t) + 4s(t + h) - s(t + 2h)}{2h}$$

Then

$$v(5) = \frac{-3(10) + 4(14.5) - 19.5}{2(1)}$$

$$= 4.25 \text{ km/s}$$

At $t = 7$, we use the central difference formulae (11.19). Therefore,

$$v(7) = \frac{s(8) - s(6)}{2h}$$

$$= \frac{25.5 - 14.5}{2} = 5.5 \text{ km/s}$$

At $t = 9$, we use the backward-difference formulae (11.18)

$$v(9) = \frac{3s(9) - 4s(8) + s(7)}{2h}$$

$$= \frac{3(32) - 4(25.5) + 19.5}{2}$$

$$= 6.75 \text{ km/s}$$

## Higher-order Derivatives

Formulae for approximating the second and higher derivatives can also be obtained from the Newton divided difference formula. The second-order derivatives are as follows:

*Central*

$$f''(x) = \frac{f(x + h) - 2f(x) + f(x - h)}{h^2} \qquad (11.20)$$

X F

$$\text{Error} = -\frac{h^2}{12} f^{(4)}(\theta)$$

*Forward*

$$f''(x) = \frac{2f(x) - 5f(x+h) + 4f(x+2h) - f(x+3h)}{h^2} \qquad (11.21)$$

$$\text{Error} = \frac{11h^2}{12} f^{(4)}(\theta)$$

*Backward*

$$f''(x) = \frac{2f(x) - 5f(x-h) + 4f(x-2h) - f(x-3h)}{h^2} \qquad (11.22)$$

$$\text{Error} = \frac{11h^2}{12} f^{(4)}(\theta)$$

**Example 11.6**

Use the table of data given in Example 11.5 to estimate acceleration at $t = 7$ s.

Acceleration is given by the second derivative of $s(t)$. Therefore

$$a(t) = s''(t) = \frac{s(t+h) - 2s(t) + s(t-h)}{h^2}$$

Therefore,

$$a(7) = \frac{25.5 - 2(19.5) + 14.5}{12}$$

$$= 1.0 \text{ km/s}^2$$

**Example 11.7**

The equation for deflection of a beam is given by

$$y''(x) - e^{x^2} = 0 \qquad y(0) = 0,\ y(1) = 0$$

Estimate, using a second-order derivative, the approximate deflections at $x = 0.25$, $0.5$, and $0.75$. Note that $y(x)$ is the deflection at $x$.

$$y''(x) = \frac{y(x+h) - 2y(x) + y(x-h)}{h^2} = e^{x^2}$$

$$h = 0.25$$

Then,

$$\frac{y(x+0.25) - 2y(x) + y(x-0.25)}{0.0625} = e^{x^2}$$

Consequently,

$$y(x + 0.25) - 2y(x) + y(x - 0.25) = 0.0625\ e^{x^2}$$

Substituting $x = 0.25$, 0.5 and 0.75 in the above equation in turn, we get

$$\begin{aligned} y(0.5) - 2y(0.25) + y(0) &= 0.0665 \\ y(0.75) - 2y(0.5) + y(0.25) &= 0.0803 \\ y(1) - 2y(0.75) + y(0.5) &= 0.1097 \end{aligned}$$

Given $y(0) = y(1) = 0$. Denoting

$$y_1 = y(0.25), \qquad y_2 = y(0.5) \qquad \text{and} \qquad y_3 = y(0.75)$$

We have

$$\begin{aligned} 0 + y_2 - 2y_1 &= 0.0665 \\ y_3 - 2y_2 + y_1 &= 0.0803 \\ -2y_3 + y_2 + 0 &= 0.1097 \end{aligned}$$

Solving these three equations for $y_1$, $y_2$ and $y_3$, we get

$$\begin{aligned} y_1 = y(0.25) &= -0.1175 \\ y_2 = y(0.5) &= -0.1684 \\ y_3 = y(0.75) &= -0.1391 \end{aligned}$$

## 11.4 DIFFERENCE TABLES

Tables 11.1 to 11.3 list difference derivatives $f'(x)$ and $f''(x)$ and associated errors for forward, backward and central difference formulae. Following notations are used:

$f_2$ denotes $f(x + 2h)$

$f_{-2}$ denotes $f(x - 2h)$

Table 11.1 Forward difference derivatives

| *Derivative* | *Formula* | *Error* |
|---|---|---|
| $f'(x_0)$ | $\frac{-f_0 + f_1}{h}$ | $-\frac{h}{2} f''(\theta)$<br>$(2e/h)^{\#}$ |
| | $\frac{-3f_0 + 4f_1 - f_2}{h}$ | $+\frac{h^2}{3} f'''(\theta)$<br>$(4e/h)$ |
| | $\frac{-11f_0 + 18f_1 - 9f_2 + 2f_3}{6h}$ | $-\frac{h^3}{4} f^{(4)}(\theta)$<br>$(20e/3h)$ |

*(Contd.)*

Table 11.1(*Contd.*)

| *Derivative* | *Formula* | *Error* |
|---|---|---|
| $f'(x_0)$ | $\dfrac{-25f_0+48f_1-36f_2+16f_3-3f_4}{12h}$ | $+\dfrac{h^4}{5}f^{(5)}(\theta)$<br>$(32e/h)$ |
| $f''(x_0)$ | $\dfrac{f_0-2f_1+f_2}{h^2}$ | $-hf'''(\theta)$<br>$(4e/h^2)$ |
|  | $\dfrac{2f_0-5f_1+4f_2-f_3}{h^2}$ | $+\dfrac{11h^2}{12}f^{(4)}(\theta)$<br>$(12e/h^2)$ |

#Roundoff error

**Table 11.2** Central difference derivatives

| *Derivative* | *Formula* | *Error* |
|---|---|---|
| $f'(x_0)$ | $\dfrac{-f_{-1}+f_1}{2h}$ | $-\dfrac{h^2}{6}f^{(3)}(\theta)$<br>$(e/h)^{\#}$ |
|  | $\dfrac{f_{-2}-8f_{-1}+8f_1-f_2}{12h}$ | $+\dfrac{h^4}{30}f^{(5)}(\theta)$<br>$(3e/2h)$ |
|  | $\dfrac{f_{-3}+9f_{-2}-45f_{-1}+45f_1-9f_2+f_3}{6h}$ | $-\dfrac{h^6}{140}f^{(7)}(\theta)$<br>$(11e/6h)$ |
| $f''(x_0)$ | $\dfrac{f_{-1}-2f_0+f_1}{h^2}$ | $-\dfrac{h^2}{12}f^{(4)}(\theta)$<br>$(4e/h^2)$ |
|  | $\dfrac{-2f_{-2}+16f_{-1}-30f_0+16f_1-f_2}{h^2}$ | $+\dfrac{h^4}{90}f^{(6)}(\theta)$<br>$(16e/3h^2)$ |

#Roundoff error

**Table 11.3** Backward difference derivatives

| *Derivative* | *Formula* | *Error* |
|---|---|---|
|  | $\dfrac{-f_{-1}+f_1}{h}$ | $\dfrac{h}{2}f''(\theta)$<br>$(2e/h)^{\#}$ |

(*Contd.*)

**Table 11.3***(Contd.)*

| *Derivative* | *Formula* | *Error* |
|---|---|---|
| $f'(x_0)$ | $\dfrac{f_{-2} - 4f_{-1} + 3f_0}{2h}$ | $\dfrac{h^4}{3} f^{(5)}(\theta)$<br>$(4e/h)$ |
| | $\dfrac{-2f_{-3} + 9f_{-2} - 18f_{-1} + 11f_0}{6h}$ | $\dfrac{h^3}{4} f^{(4)}(\theta)$<br>$(20e/3h)$ |
| | $\dfrac{3f_{-4} - 16f_{-3} + 36f_{-2} - 48f_{-1} + 25f_0}{12h}$ | $\dfrac{h^3}{5} f^{(5)}(\theta)$<br>$(32e/h)$ |
| $f''(x_0)$ | $\dfrac{f_{-2} - 2f_{-1} + f_0}{h^2}$ | $hf^{(3)}(\theta)$<br>$(4e/h^2)$ |
| | $\dfrac{-f_{-3} + 4f_{-2} - 5f_{-1} + 2f_0}{h^2}$ | $\dfrac{11h^2}{12} f^{(4)}(\theta)$<br>$(12e/h^2)$ |

#Roundoff error

## 11.5 RICHARDSON EXTRAPOLATION

Richardson extrapolation is based on a model for the error in a numerical process. This is used to improve the estimates of numerical solutions. Let us assume

$$x_k = x^* + M\,h^n \tag{11.23}$$

$x_k$ is the $k$th estimate of solution $x^*$ and $M\,h^n$ is the error term. Let us now replace $h$ by $rh$ and obtain another estimate for $x^*$.

$$x_{k+1} = x^* + M\,r^n\,h^n \tag{11.24}$$

Multiplying Eq. (11.23) by $r^n$ and solving for $x^*$, we get

$$\boxed{x^* = x_R = \frac{x_{k+1} - r^n x_k}{1 - r^n}} \tag{11.25}$$

This is known as *Richardson extrapolation estimate*. Note that the error term has been eliminated.

This concept can be extended to the estimation of derivatives discussed so far. Using this, we can obtain a higher-order formula from a lower-order formula, thus improving the accuracy of the estimates. This process is known as *extrapolation*. Let us consider the three-point central difference formula with its error term.

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - \frac{h^2}{6} f'''(\theta)$$

$$= D(h) - \frac{h^2}{6} f'''(\theta) \tag{11.26}$$

where $D(h)$ is the estimate obtained using $h$ as step size. Note that $f'(x)$ is the exact solution which is usually approximated by $D(h)$. If we remove the error term, then we can obtain a better approximation. Now, let us obtain another approximation for $f'(x)$ by replacing $h$ by $rh$. Thus,

$$f'(x) = \frac{f(x+rh) - f(x-rh)}{2hr} - \frac{h^2 r^2}{6} f''(\theta)$$

$$= D(rh) - \frac{h^2 r^2}{6} f''(\theta) \tag{11.27}$$

We can eliminate the error term by multiplying Eq. (11.26) by $r^2$ and subtracting it from Eq. (11.27). The result would be

$$f'(x) = \frac{D(rh) - r^2\, D(h)}{1 - r^2} \tag{11.28}$$

This would give a better estimate of $f'(x)$ as we have eliminated the error term $h^2$. For $r = 2$, Eq. (11.28) becomes

$$f'(x) = \frac{f(x-2h) - 8f(x-h) + 8f(x+h) - f(x+2h)}{12h} \tag{11.29}$$

Note that this is a five-point central difference formula which contains error only in the order of $h^4$. We can repeat this process further to eliminate the error term containing $h^4$ and so on.

One of the most common choices of $r$ is 0.5. Letting $r = 1/2$, Eq. (11.28) becomes

$$f'(x) = \frac{f(x-h) - 8(x-h/2) + 8f(x+h/2) - f(x+h)}{6h} \tag{11.30}$$

Note that the use of this formula depends on the availability of function values at $x \pm h/2$ points. This will be a restriction when Richardson's extrapolation technique is applied to tabulated functions.

### Example 11.8

Show that, using the data given below, Richardson's extrapolation technique can provide better estimates for derivatives.

| $x$ | $-0.5$ | $-0.25$ | $0$ | $0.25$ | $0.5$ | $0.75$ | $1.0$ | $1.25$ | $1.5$ |
|---|---|---|---|---|---|---|---|---|---|
| $f(x) = e^x$ | 0.6065 | 0.7788 | 1.0000 | 1.2840 | 1.6487 | 2.1170 | 2.7183 | 3.4903 | 4.4817 |

Let us estimate $f'(x)$ at $x = 0.5$ and assume $h = 0.5$ and $r = 1/2$. Then, using three-point central formula, we have

$$D(h) = D(0.5) = \frac{f(1.0) - f(0)}{2 \times 0.5} = 1.7183$$

$$D(rh) = D(0.25) = \frac{f(0.75) - f(0.25)}{0.5} = 1.666$$

$$f'(x) = \frac{D(rh) - r^2 D(h)}{1 - r^2}$$

Therefore,

$$f'(0.5) = \frac{1.6660 - 0.25(1.7183)}{0.75}$$

$$= 1.6486$$

Note that the correct answer is 1.6487. The result is much better than the results obtained using three-point formula with $h = 0.5$ and $h = 0.25$.

Now, let us take $r = 2$. Again using the same three-point central formula.

$$D(rh) = D(1.0) = \frac{f(1.5) - f(-0.5)}{(2)(0.5)(2)} = 1.9376$$

$$f(x) = \frac{1.9376 - 4(1.7183)}{-3} = 1.6452$$

This shows that the estimate with $r = 1/2$ is better than the estimate with $r = 2$.

## 11.6 SUMMARY

In this chapter, we have seen how numerical differentiation techniques may be used to obtain the derivative of continuous as well as tabulated functions. We have used forward, backward and central difference quotients to obtain derivative equations. We have also seen how Richardson extrapolation is used to improve the estimates of numerical solutions.

The discussions in this chapter bring out the following points:

- If we are given $n + 1$ data points equally spaced, the interpolating polynomial will be of order $n$, and the $n$th derivative will be the highest that can be obtained.

- Better approximation of derivatives can be achieved by using more points in the formula.
- For a given number of data points, the central difference formula is more accurate than their forward or backward counter parts.
- Roundoff error grows when $h$ gets small. We will always face the *step-size dilemma*. One way to overcome this problem is to use a formula of higher order so that a large value of $h$ will produce the desired accuracy.
- The problem becomes more pronounced when working with experimental data which contain not only roundoff errors but also measurement errors. In such cases, we should first fit a curve to the data by using least-squares technique and compute derivatives for the curve.

## Key Terms

*Backward difference derivative*
*Backward difference quotient*
*Central difference derivative*
*Central difference quotient*
*Difference tables*
*Extrapolation*
*Five-point formula*
*Forward difference derivative*
*Forward difference quotient*
*Richardson extrapolation*
*Two-point formula*
*Three-point formula*

## REVIEW QUESTIONS

1. What is numerical differentiation?
2. Why do we need to use numerical techniques to obtain the estimates of function derivatives?
3. What are the three primitive numerical differentiation formulae? Compare their truncation errors.
4. What is three-point formula? How is it different from the two-point formula? Illustrate the difference using geometric interpretations.
5. Derive the five-point central difference formula

$$f'(x) = \frac{-f(x+2h) + 8f(x+h) - 8f(x-h) + f(x-2h)}{12h}$$

   Also estimate the order of truncation error.
6. Describe the effect of step size $h$ on
   (a) truncation error,
   (b) roundoff error, and
   (c) total error.
7. Using Taylor's expansion, derive a formula for computing second derivative of a function.
8. Derive a three-point difference formula for estimating the first derivative of a tabulated function.

9. Derive a formula to estimate the second derivative of a tabulated function.
10. What is Richardson extrapolation? How does it improve the estimates of derivatives?

## [illegible]

1. Estimate the first derivative of $f(x) = \ln x$ at $x = 1$ using the first order.
   (a) first-order forward difference formula,
   (b) first-order backward difference formula, and
   (c) second-order central difference formula.
   Compare the results with the exact value 1.
2. Estimate the first derivative of $f(x) = \ln x$ using the five-point central difference formula. How does the result compare with the results obtained in Exercise 1.
3. Compute the approximate first derivatives of $f(x) = \cos x$ at $x = 0.75$ radians at increasing values of $h$ from 0.01 to 0.05 with a step size of 0.005 (using four decimal digits). Analyze the variations of error in each step.
4. Apply the three-point central difference formula to obtain estimates of the first derivatives of the following functions at $x = 1$ with $h = 0.01$. Compare the results with true values.
   (a) $\cosh x$
   (b) $\exp(x) \sin x$
   (c) $\ln(1 + x^2)$
   (d) $x^2 + 2x + 1$
   (e) $\dfrac{1}{1+x^2}$
5. For each of the following functions
   (a) $\cos x$ $\quad x = 1.5$
   (b) $\exp(x/2)$ $\quad x = 2$
   (c) $\dfrac{1}{1+x^2}$ $\quad x = 1$

   estimate the size of $h$ that will minimize total error when using the three-point central difference formula.
6. Estimate the first derivatives of the functions given in Exercise 5 at the indicated points using the optimum size $h$ obtained.
7. Use the three-point formula to estimate the second derivatives of the functions given in Exercise 4 at $x = 0.5$ with $h = 0.01$.
8. Given below the table of function values of $f(x) = \sin h(x)$. Estimate the second derivatives of $f(x)$ at $x = 1.2$, 1.3 and 1.4 using a suitable formula.

| $x$ | 1.1 | 1.2 | 1.3 | 1.4 | 1.5 |
|---|---|---|---|---|---|
| $f(x)$ | 1.3356 | 1.5095 | 1.6983 | 1.9043 | 2.1293 |

9. Current through a capacitor is given by

$$I(t) = C\,\frac{dv}{dt} = Cv'(t)$$

where $v(t)$ is the voltage across the capacitor at time $t$ and $C$ is the capacitance value of the capacitor. Estimate the current through the capacitor at $t = 0.5$ using the two-point forward formula with a step size $h = 0.2$. Assume the following:

$$v(t) = (t + 0.1)\,e^{\sqrt{t}} \text{ volts}$$
$$C = 2\,F$$

10. Using the function in Exercise 8, estimate the first derivative at $x = 1.3$ with $h = 0.1$ using the three-point centre formula. Compute an improved estimate using Richardson extrapolation. Exact value of $f'(x) = \cosh(1.3) = 1.9709$.
11. Evaluate the first derivative at $x = -3$ and $x = 0$ of the following table function:

| $x$ | $-3$ | $-2$ | $-1$ | 0 | 1 | 2 | 3 |
|---|---|---|---|---|---|---|---|
| $y$ | $-33$ | $-12$ | $-3$ | 0 | 3 | 12 | 33 |

12. Compute the first derivative for the following table of data at $x = 0.75$, 1.00 and 1.25. Use $h = 0.05$ and 0.1.

| $x$ | 0.5 | 0.7 | 0.9 | 1.1 | 1.3 | 1.5 |
|---|---|---|---|---|---|---|
| $y$ | 1.48 | 1.64 | 1.78 | 1.89 | 1.96 | 1.00 |

Compare the results with $h = 0.05$ and $h = 0.1$. Comment on the differences, if any.

13. The following table gives the velocity of an object at various points in time

| Time (seconds) | 1 | 1.2 | 1.6 | 1.8 | 2.2 | 2.4 | 2.8 | 3.0 |
|---|---|---|---|---|---|---|---|---|
| Velocity (m/sec) | 9.0 | 9.5 | 10.2 | 11.0 | 13.2 | 14.7 | 18.7 | 22.0 |

Find the acceleration of the object at $T = 2.0$ seconds. Assume a suitable value for $h$.

14. The distances travelled by a vehicle at intervals of 2 minutes are given as follows:

| Time (seconds) | 0 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 |
|---|---|---|---|---|---|---|---|---|---|
| Distance (km) | 0 | 0.25 | 1 | 2.2 | 4 | 6.5 | 8.5 | 11 | 13 |

Evaluate the velocity and acceleration of the vehicle at $T = 5$, 10 and 13 seconds.

## POGRAMMING PROJECTS

1. Write a program that will read on the values of $x$ and $f(x)$, compute approximations to $f'(x)$ and $f''(x)$, and output $x, f(x), f'(x)$ and $f''(x)$ in four columns.
2. Write a program that will compute the total error at increasing values of $h$ at regular steps and then estimate that value of $h$ for which the total error is minimum. Assume a formula of your choice.
3. Write a program to evaluate a given function at various points of interest and estimate its first and second derivatives at any specified point.

CHAPTER 12

# Numerical Integration

## 12.1 NEED AND SCOPE

Like numerical differentiation, we need to seek the help of numerical integration techniques in the following situations:

1. Functions do not possess closed form solutions. Example:

$$f(x) = C \int_0^x e^{-t^2} \mathrm{d}t$$

2. Closed form solutions exist but these solutions are complex and difficult to use for calculations.
3. Data for variables are available in the form of a table, but no mathematical relationship between them is known, as is often the case with experimental data.

We know that a definite integral of the form

$$I = \int_a^b f(x)\,\mathrm{d}x \tag{12.1}$$

can be treated as the area under the curve $y = f(x)$, enclosed between the limits $x = a$ and $x = b$. This is graphically illustrated in Fig. 12.1. The problem of integration is then simply reduced to the problem of finding the shaded area.

One simple approach is to plot the function on a graph paper containing grids and find the area under the curve using the number grids covered under the desired boundaries. The accuracy of this rough estimate can be improved by using finer grids.

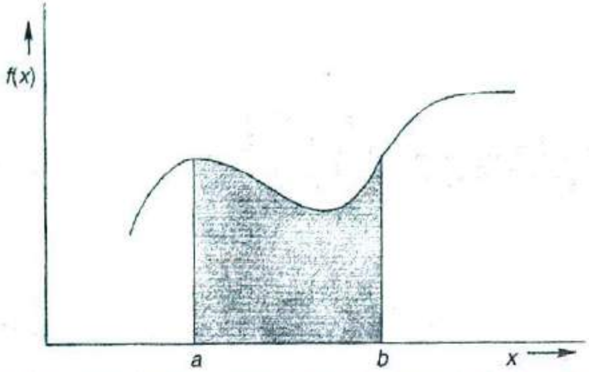


Fig. 12.1 Graphical representation of integral of a function

Although the grid method and other such graphical approaches can provide us rough estimates, they are cumbersome and time-consuming and the final results are far from satisfactory limits. A better alternative approach could be to use a technique that uses simple arithmetic operations to compute the area. Such an approach, if necessary, can be easily implemented on a computer. This approach is called *numerical integration* or *numerical quadrature*. Numerical integration techniques are similar in spirit to the graphical methods. Both of them use the concept of "summation" to find the area.

Numerical integration methods use an interpolating polynomial $p_n(x)$ in the place of $f(x)$. Thus

$$I = \int_a^b f(x)\,\mathrm{d}x = \int_a^b p_n(x)\,\mathrm{d}x \tag{12.2}$$

We know that the polynomial $p_n(x)$ can be easily integrated analytically. Equation (12.2) can be expressed in summation form as follows:

$$\int_a^b p_n(x)\,\mathrm{d}x = \sum_{i=0}^{n} w_i\, p_n(x_i) \tag{12.3}$$

where $a = x_0 < x_1 < \ldots < x_n = b$

Since $p_n(x)$ coincides with $f(x)$ at all the points $x_i$, $i = 0, 1, \ldots n$, we can say that,

$$\boxed{I = \int_a^b f(x)\,\mathrm{d}x \approx \sum_{i=0}^{n} w_i\,(x_i)} \tag{12.4}$$

The values $x_i$ are called *sampling points* or *integration nodes* and the constants $w_i$ are called *weighting coefficients* or simply *weights*.

Equation (12.4) provides the basic integration formula that will be extensively used in this chapter. Note that the interpolation polynomial

$p_n(x)$ was used only to derive the formula (12.4) and will not be used in the computation directly. Only the actual function values at sample points are used in numerical computation.

There are various methods of selecting the location and number of sampling points. There is a set of methods known as *Newton-Cotes rules* in which the sampling points are equally spaced. Another set of methods called *Gauss-Legendre rules,* uses sampling points that are not equally spaced, but are designed to provide improved accuracy. In this chapter, we discuss these two sets of methods in detail. We also discuss a method known as *Romberg integration* that is designed to improve the estimates of Newton-Cotes formulae.

In general, numerical integration methods yield much better results compared to the numerical differentiation methods discussed in the previous chapter. This is due to the fact that the errors introduced in separate subintervals tend to cancel each other. However, the estimates are still approximate and, therefore, we also consider the magnitude of errors in each of the methods discussed here.

## 12.2 NEWTON-COTES METHODS

Newton-Cotes formula is the most popular and widely used numerical integration formula. It forms the basis for a number of numerical integration methods known as *Newton-Cotes methods.*

The derivation of Newton-Cotes formula is based on polynomial interpolation. As pointed our earlier, an $n$th degree polynomial $p_n(x)$ that interpolates the values of $f(x)$ at $n + 1$ evenly spaced points can be used to replace the integrand $f(x)$ of the integral

$$I = \int_a^b f(x)\,dx$$

and the resultant formula is called $(n + 1)$ point *Newton-Cotes formula*. If the limits of integration $a$ and $b$ are in the set of interpolating points $x_i$, $i = 0, 1, \dots n$, then the formula is referred to as *closed form*. If the points $a$ and $b$ lie beyond the set of interpolating points, then the formula is termed *open form*. Since open form formula is not used for definite integration, we consider here only the closed form methods. They include:

1. Trapezoidal rule (two-point formula)
2. Simpson's 1/3 rule (three-point formula)
3. Simpson's 3/8 rule (four-point formula)
4. Boole's rule (five-point formula)

All these rules can be formulated using either Newton or Lagrange interpolation polynomial for approximating the function $f(x)$. We use here the Newton-Gregory forward formula (Eq. (9.20)) which is given below:

$$p_n(s) = f_0 + \Delta f_0 s + \frac{\Delta^2 f_0}{2!} s(s-1) + \frac{\Delta^3 f_0}{3!} s(s-1)(s-2) + \ldots$$

$$= T_0 + T_1 + T_2 + \ldots + T_n \tag{12.5}$$

where

$$s = (x - x_0)/h$$

and

$$h = x_{i+1} - x_i$$

## 12.3 TRAPEZOIDAL RULE

✓ NUB

The trapezoidal rule is the first and the simplest of the Newton-Cotes formulae. Since it is a two-point formula, it uses the first order interpolation polynomial $p_1(x)$ for approximating the function $f(x)$ and assumes $x_0 = a$ and $x_1 = b$. This is illustrated in Fig. 12.2. According to Eq. (12.5), $p_1(x)$ consists of the first two terms $T_0$ and $T_1$. Therefore, the integral for trapezoidal rule is given by

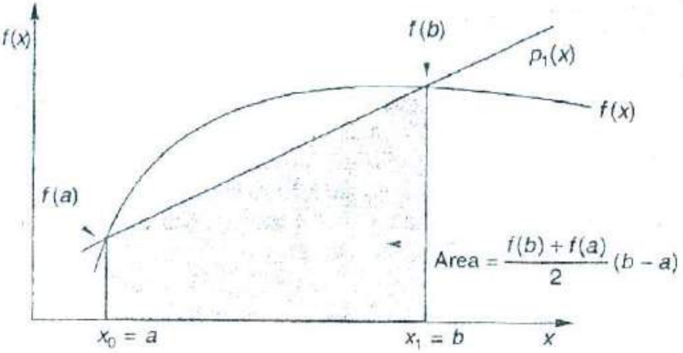


Fig. 12.2 Representation of trapezoidal rule

$$I_t = \int_a^b (T_0 + T_1)\mathrm{d}x$$

$$= \int_a^b T_0 \, \mathrm{d}x + \int_a^b T_1 \, \mathrm{d}x = I_{t1} + I_{t2}$$

Since $T_i$ are expressed in terms of $s$, we need to use the following transformation:

$$\mathrm{d}x = h \times \mathrm{d}s$$

$$x_0 = a, \quad x_1 = b \quad \text{and} \quad h = b - a$$

At $\quad x = a, \quad s = (a - x_0)/h = 0$

At $\quad x = b, \quad s = (b - x_0)/h = 1$

Then,

$$I_{t1} = \int_a^b T_0 \, \mathrm{d}x = \int_0^1 hf_0 \, \mathrm{d}x = hf_0$$

$$I_{t2} = \int_a^b T_1 \, \mathrm{d}x = \int_0^1 \Delta f_0 \; sh \, \mathrm{d}s = h \frac{\Delta f_0}{2}$$

Therefore,

$$I_t = h\left[f_0 + \frac{\Delta f_0}{2}\right] = h\left[\frac{f_0 + f_1}{2}\right]$$

Since $f_0 = f(a)$ and $f_1 = f(b)$, we have

$$\boxed{I_t = h\,\frac{f(a) + f(b)}{2} = (b - a)\,\frac{f(a) + f(b)}{2}} \qquad (12.6)$$

Note that the area is the *product of width of the segment* $(b - a)$ *and average height of the points f(a) and f(b).*

## Error Analysis

Since only the first two terms of eq. (12.5) are used for $I_t$, the term $T_2$ becomes the remainder and, therefore, the truncation error in trapezoidal rule is given by

$$E_{tt} = \int_a^b T_2 \, \mathrm{d}x = \frac{f''(\theta_s)}{2} \int_0^1 s(s-1)h \cdot \mathrm{d}s$$

$$= \frac{f''(\theta_s)\,h}{2}\left[\frac{s^3}{3} - \frac{s^2}{2}\right]_0^1 = -\frac{f''(\theta_s)}{12}\,h$$

Since $\mathrm{d}x/\mathrm{d}s = h$,

$$f''(\theta_s) = h^2 f''(\theta_x),$$

we obtain

$$\boxed{E_{tt} = -\frac{h^3}{12} f''(\theta_x)} \qquad (12.7)$$

where $a < \theta_x < b$

**Example 12.1** NUB

Evaluate the integral

$$I = \int_a^b (x^3 + 1)\,\mathrm{d}x$$

for the intervals (a) (1, 2) and (b) (1, 1.5)
Also estimate truncation error in each case and compare the results with the exact answer.

*Case a*

$$a = 1, b = 2$$
$$h = 1$$
$$I_t = \frac{b-a}{2}\,[f(a) + f(b)]$$
$$= \frac{1}{2}\,(2 + 9) = 5.5$$
$$|E_{tt}| \le \frac{h^3}{12} \max_{1 \le x \le 2} |f''(x)|$$
$$f''(x) = 6x$$
$$\max_{1 \le x \le 2} |f''(x)| = f''(2) = 12$$

Therefore,

$$|E_{tt}| \le \frac{h^3}{12}\, f''(2) = 1$$
$$I_{\text{exact}} = 9.75$$
$$\text{True error} = I_t - I_{\text{exact}} = 0.75$$

Note that the error bound is an overestimate of the true error.

*Case b*

$$a = 1, b = 1.5$$
$$h = 0.5$$
$$I_t = \frac{0.5}{2}\,[f(1) + f(1.5)] = 1.59375$$
$$|E_{tt}| = \frac{(0.5)^3}{12}\, f''(1.5) = 0.09375$$
$$I_{\text{exact}} = 1.515625$$
$$\text{True error} = 0.078125$$

## Composite Trapezoidal Rule

If the range to be integrated is large, the trapezoidal rule can be improved by dividing the interval $(a, b)$ into a number of small intervals and applying the rule discussed above to each of these subintervals. The sum of areas of all the subintervals is the integral of the interval $(a, b)$. This is known as *composite* or *multisegment approach*. This is illustrated in Fig. 12.3.

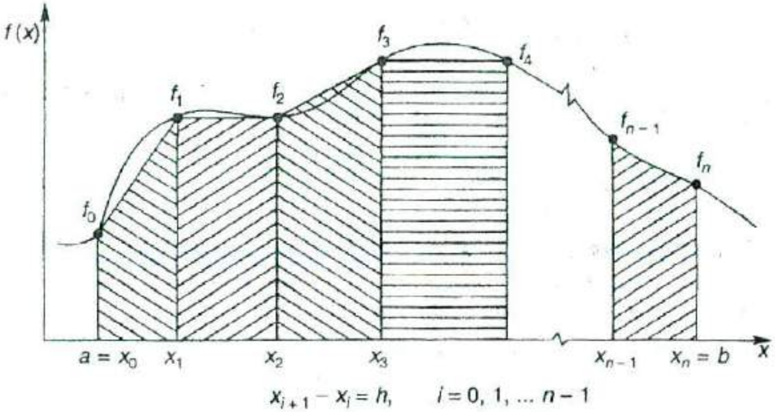


**Fig. 12.3** Multisegment trapezoidal rule

As seen in Fig. 12.3, there are $n + 1$ equally spaced sampling points that create $n$ segments of equal width $h$ given by

$$h = \frac{b-a}{n}$$

$$x_i = a + ih, \qquad i = 0, 1, \dots, n$$

From Eq. (12.6), area of the subinterval with the nodes $x_{i-1}$ and $x_i$, is given by

$$I_i = \int_{x_{i-1}}^{x_i} p_1(x)\,\mathrm{d}x = \frac{h}{2}\,[f(x_{i-1}) + (x_i)]$$

The total area of all the $n$ segments is

$$I_{ct} = \sum_{i=1}^{n} \frac{h}{2}\,[f(x_{i-1}) + f(x_i)]$$

$$= \frac{h}{2}\,[f(x_0) + f(x_1)] + \frac{h}{2}\,[f(x_1) + f(x_2)]$$

$$+ \dots + \frac{h}{2}\,[f(x_{n-1}) + f(x_n)]$$

Denoting $f_i = f(x_i)$ and regrouping the terms, we get

$$\boxed{I_{ct} = \frac{h}{2}\left[f_0 + 2\sum_{i=1}^{n-1} f_i + f_n\right]} \qquad (12.8)$$

Equation (12.8) is the general form of trapezoidal rule and is known as *composite trapezoidal rule*. Equation (12.8) can also be expressed as follows:

$$I_{ct} = \frac{h}{2}\,[f(a) + f(b)] + h\,\sum_{i=1}^{n-1} f(a + ih)$$

Similarly, we can estimate the error in the composite trapezoidal rule by adding the errors of individual segments. Thus

$$E_{ctt} = -\frac{h^3}{12}\sum_{i=1}^{n} f''(\theta_i) \tag{12.9}$$

We know that f ¢¢($q_i$) is the second derivative at $q_i$, $x_{i-1} < q_i < x_i$. If the maximum absolute value of the second derivative in the interval $(a, b)$ is $F$, then we can say that the truncation error is

$$\boxed{E_{ctt} \le \frac{h^3}{12}\,nF = \frac{(b-a)^3}{12n^2}\,F} \tag{12.10}$$

Theoretically, we can say that it is always possible to increase accuracy by taking more and more segments. Unfortunately, this does not happen always. When the number of segments increases, error due to rounding off increases.

### Example 12.2

Compute the integral

$$\int_{-1}^{1} e^{x}\,\mathrm{d}x$$

using composite trapezoidal rule for (a) $n = 2$ and (b) $n = 4$.

*Case a* $\quad n = 2$

$$h = \frac{b-a}{2} = \frac{2}{2} = 1$$

$$I_{ct} = \frac{h}{2}\,[f(a) + f(b)] + h\,\sum_{i=1}^{n-1} f(a + ih)$$

$$= \frac{1}{2}\,[\exp(-1) + \exp(1)] + \exp(0)$$

$$= 2.54308$$

*Case b* $\quad n = 4$

$$h = \frac{b-a}{4} = 0.5$$

$$= \frac{0.5}{2} + [\exp(-1) + \exp(1)] + [\exp(-0.5) + \exp(0) + \exp(0.5)]\ 0.5$$

$$= 2.39917$$

Note that $I_{\text{exact}} = 2.35040$ and $n = 4$ gives better results.

## Program TRAPE1

Trapezoidal rule is a simple algorithm and can be implemented by a few FORTRAN statements as shown in the program TRAPE1. Note that the major computation is done by just one statement in a DO loop. This statement calls a function subprogram to evaluate the given function at a specified value of $x$. We can use this program to integrate any function by simply changing the function definition statement

```
                    F = 1 - EXP(-X/2.0)
```

```
* -------------------------------------------------------- *
      PROGRAM TRAPE1
* -------------------------------------------------------- *
* Main program                                             *
*     This program integrates a given function
*     using the trapezoidal rule                           *
* -------------------------------------------------------- *
* Functions invoked                                        *
*     F                                                    *
* -------------------------------------------------------- *
* Subroutines used                                         *
*     NIL                                                  *
* Variables used                                           *
*     A - Lower limit of integration                       *
*     B - Upper limit of integration                       *
*     H - Segment width                                    *
*     N - Number of segments                               *
*   ICT - Value of integral                                *
* -------------------------------------------------------- *
* Constants used                                           *
*     NIL                                                  *
* -------------------------------------------------------- *

      INTEGER N
      REAL A, B, H, SUM, ICT
      EXTERNAL F

      WRITE(*, *) 'Give initial value of X'
      READ(*, *) A
      WRITE(*, *) 'Give final value of X'
      READ(*, *) B
      WRITE(*, *) 'What is the segment width?'
      READ(*, *) H
      N = (B-A)/H

      SUM = (F(A) + F(B))/2.0
      DO 10 I = 1, N-1
         SUM = SUM + F(A+I*H)
10    CONTINUE

      ICT = SUM * H

      WRITE(*, *)
      WRITE(*, *) 'INTEGRATION BETWEEN', A,' AND', B
      WRITE(*, *)
      WRITE(*, *) 'WHEN H =', H, ' IS', ICT
      WRITE(*, *)

      STOP
      END
* ------------- End of main TRAPE1 ---------------- *
* ------------------------------------------------- *
* Function subprogram F (X)                         *
* ------------------------------------------------- *

      REAL FUNCTION F(X)
      REAL X

      F = 1-EXP(-X/2.0)

      RETURN
      END
* -------------- End of function F(X) -------------- *
```

***Test Run Results*** Test run results shown below give the value of integration of the equation

$$f(x) = 1 - e^{-x/2}$$

from 0.0 to 10.0.

```
      Give initial value of X
      0.0
      Give final value of X
      10.0
      What is the segment width?
      0.5

      INTEGRATION BETWEEN .0000000 AND 10.0000000

      WHEN H = 5.000000E-001 IS 8.0031400

      Stop - Program terminated.
```

## 12.4 SIMPSON'S 1/3 RULE

Another popular method is Simpson's 1/3 rule. Here, the function $f(x)$ is approximated by a second-order polynomial $p_2(x)$ which passes through three sampling points as shown in Fig. 12.4. The three points include the end points $a$ and $b$ and a midpoint between them, i.e., $x_0 = a$, $x_2 = b$ and $x_1 = (a + b)/2$. The width of the segments $h$ is given by

$$h = \frac{b-a}{2}$$

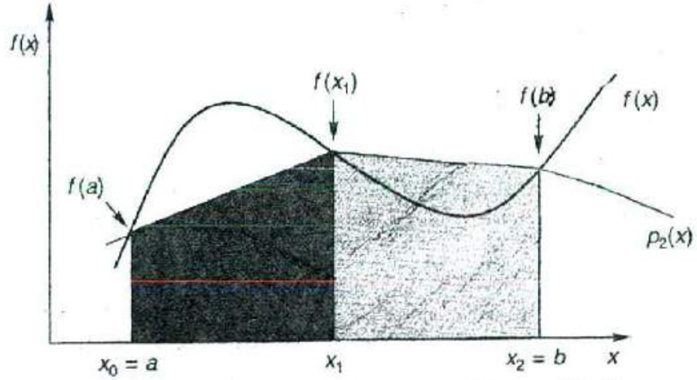


Fig. 12.4 Representation of Simpson's Three-point rule

The integral for Simpson's 1/3 rule is obtained by integrating the first three terms of equation (12.5), i.e.,

$$I_{s1} = \int_a^b p_2(x)\,dx = \int_a^b (T_0 + T_1 + T_2)\,dx$$

$$= \int_a^b T_0\,dx + \int_a^b T_1\,dx + \int_a^b T_2\,dx$$

$$= I_{s11} + I_{s12} + I_{s13}$$

where

$$I_{s11} = \int_a^b f_0\,dx$$

$$I_{s12} = \int_a^b \Delta f_0\, s\,dx$$

$$I_{s13} = \int_a^b \frac{\Delta^2 f_0}{2}\, s(s-1)\,dx$$

We know that $dx = h \times ds$ and $s$ varies from 0 to 2 (when $x$ varies from $a$ to $b$). Thus,

$$I_{s11} = \int_0^2 f_0 h \, ds = 2hf_0$$

$$I_{s12} = \int_0^2 \Delta f_0 sh \, ds = 2h\Delta f_0$$

$$I_{s13} = \int_0^2 \frac{\Delta^2 f_0}{2} s(s-1)h \, ds = \frac{h}{3}\Delta^2 f_0$$

Therefore,

$$I_{s1} = h\left[ sf_0 + 2\Delta f_0 + \frac{\Delta^2 f_0}{3} \right] \tag{12.11}$$

Since $\Delta f_0 = f_1 - f_0$ and $\Delta^2 f_0 = f_2 - 2f_1 + f_0$, equation (12.11) becomes

$$\boxed{I_{s1} = \frac{h}{3}\,[f_0 + 4f_1 + f_2] = \frac{h}{3}\,[f(a) + 4f(x_1) + f(b)]} \tag{12.12}$$

This equation is called *Simpson's 1/3 rule*. Equation (12.12) can also be expressed as

$$I_{s1} = (b-a)\frac{f(a) + 4f(x_1) + f(b)}{6}$$

This shows that the area is given by *the product of total width of the segments and weighted average of heights $f(a)$, $f(x_1)$ and $f(b)$.*

## Error Analysis

Since we have used only the first three terms of Eq. (12.5), the truncation error is given by

$$E_{ts1} = \int_a^b T_3 \, dx$$

$$= \frac{f'''(\theta_s)}{6} \int_0^2 s(s-1)(s-2)h \, ds$$

$$= \frac{f'''(\theta_s)}{6} \left[ \frac{s^4}{4} - s^3 + s^2 \right]_0^2$$

Since the third-order error term turns out to be zero, we have to consider the next higher term for the error. Therefore,

$$E_{ts1} = \int_a^b T_4 \, dx$$

$$= \frac{f^{(4)}(\theta_s)}{4!}\int_0^2 s(s-1)(s-2)(s-3)h\,\mathrm{d}s$$

$$= \frac{h \times f^{(4)}(\theta_s)}{24}\left[\frac{s^5}{5} - \frac{6s^4}{4} + \frac{11s^3}{3} - \frac{6s^2}{2}\right]_0^2$$

$$= -\frac{hf^4(\theta_s)}{90}$$

Since $f^4(\theta_s) = h^4 f^{(4)}(\theta_x)$, we obtain

$$\boxed{E_{ts1} = -\frac{h^5}{90} f^{(4)}(\theta_x)} \tag{12.13}$$

where $a < \theta_x < b$. It is important to note that Simpson's 1/3 rule is exact up to degree 3, although it is based on quadratic equation.

NUB

### Example 12.3

Evaluate the following integrals using Simpson's 1/3 rule

(a) $\int_{-1}^{1} e^x\,\mathrm{d}x$ (b) $\int_0^{\pi} \sqrt{\sin x}\,\mathrm{d}x$

*Case (a)*

$$I = \int_{-1}^{1} e^x\,\mathrm{d}x$$

$$I_{s1} = \frac{h}{3}\,[f(a) + f(b) + 4f(x_1)]$$

$$h = \frac{b-a}{2} = 1$$

$$f(x_1) = f(a + b)$$

Therefore,

$$I_{s1} = \frac{e^{-1} + 4e^0 + e^{+1}}{3} = 2.36205$$

(Note that $I_{s1}$ gives better estimate than $I_{ct}$ when $n = 2$. This is because $I_{s1}$ uses quadratic equation while $I_{ct}$ uses a linear one)

*Case (b)*

$$I = \int_0^{\pi/2} \sqrt{\sin(x)}\,\mathrm{d}x = \pi/4$$

$$I_{s1} = \frac{\pi}{12}\,[f(0) + 4f(\pi/4) + f(\pi/2)]$$

$$= 0.2617993(0 + 3.3635857 + 1)$$

$$= 1.1423841$$

## Composite Simpson's 1/3 rule

Similar to the composite trapezoidal rule, we can construct a composite Simpson's 1/3 rule to improve the accuracy of the estimate of the area. Here again, the integration interval is divided into $n$ number of segments of equal width, where $n$ is an even number. Then the step size is

$$h = \frac{b-a}{n}$$

As usual, $x_i = a + ih$, $i = 0, 1, \dots n$. Now, we can apply Eq. (12.12) to each of the $n/2$ pairs of segments or subintervals $(x_{2i-2}, x_{2i-1})$, $(x_{2i-1}, x_{2i})$. This gives

$$I_{cs1} = \frac{h}{3} \sum_{i=1}^{n/2} [f(x_{2i-2}) + 4f(x_{2i-1}) + f(x_{2i})]$$

$$= \frac{h}{3} \, [f(a) + 4f_1 + 2f_2 + 4f_3 + \dots 2f_{n-2} + 4f_{n-1} + f(b)]$$

On regrouping terms, we get

$$\boxed{I_{cs1} = \frac{h}{3}\left[f(a) + 4\sum_{i=1}^{n/2} f(x_{2i-1}) + 2\sum_{i=1}^{(n/2)-1} f(x_{2i}) + f(b)\right]} \qquad (12.14)$$

An analysis similar to the error analysis of composite trapezoidal rule can be performed to obtain the error due to truncation in composite Simpson's 1/3 rule.

$$\boxed{|E_{tcs1}| \le \frac{h^5}{180} nF = \frac{(b-a)^5}{180n^4} F} \qquad (12.15)$$

where $F$ is the maximum absolute value of the fourth derivative of $f(x)$ in the interval $(a, b)$.

### Example 12.4

Compute the integral

$$\int_0^{\pi/2} \sqrt{\sin(x)} \, dx$$

applying Simpson's 1/3 rule for $n = 4$ and $n = 6$ with an accuracy to five decimal places.

The composite Simpson's 1/3 rule is given by

$$I_{cs1} = \frac{h}{3}\left[(f(x_0) + f(x_a)) + 4\sum_{i=1}^{n/2} f(x_{2i-1}) + 2\sum_{i=1}^{n/2-1} f(x_{2i})\right]$$

*For $n = 4$, $h = \pi/8$*

There are five sampling points given by $x_k = k\pi/8$, $k = 0, 1, ..., 4$. Substituting the values of $x_k$ in the composite rule, we get,

$$I_{cs1} = \frac{\pi}{24}\,[f(0) + f(\pi/2) + 4f(\pi/8) + 4f(3\pi/8) + 2f(\pi/4)]$$

$$= \frac{\pi}{24}\,[0 + 1.0 + 4(0.61861 + 0.96119) + 2\,(0.84090)]$$

$$= 1.17823$$

*For $n = 6$, $h = \pi/12$*

There are seven sampling points given by $x_k = k\pi/12$, $k = 0, 1, ..., 6$ Substituting these values in the above equation, we get

$$I_{cs1} = \frac{\pi}{36}[0 + 1.0 + 4(0.50874 + 0.84090 + 0.98282) + 2(0.70711 + 0.93060)$$

$$= 1.18728$$

## Program SIMS1

Program SIMS1 integrates a given function using the composite Simpson's 1/3 rule. Note that, unlike TRAPE1 which requests for segment width $h$, SIMS1 requests for number of segments $n$. Remember, $n$ should be even. This program also uses a function subprogram which can be easily replaced for any other function without modifying the main program.

```
* ------------------------------------------------------- *
      PROGRAM SIMS1
* ------------------------------------------------------- *
* Main program                                            *
*   This program integrates a given function              *
*   using the Simpson's 1/3 rule                          *
* ------------------------------------------------------- *
* Functions invoked                                       *
*   F                                                     *
* ------------------------------------------------------- *
* Subroutines used                                        *
*   NIL                                                   *
* ------------------------------------------------------- *
* Variables used                                          *
*   A - Lower limit of integration                        *
*   B - Upper limit of integration                        *
*   H - Segment width                                     *
*   N - Number of segments                                *
* ICS - Value of the integral                             *
* ------------------------------------------------------- *
* Constants used                                          *
*   NIL                                                   *
* ------------------------------------------------------- *
```

```
      INTEGER N,M
      REAL A,B,H,SUM,ICS,X,F1,F2,F3
      EXTERNAL F

      WRITE (*,*) 'Initial value of X'
      READ (*,*) A
      WRITE(*,*) 'Final value of X'
      READ(*,*) B
      WRITE(*,*) 'Number of segments (EVEN number)'
      READ(*,*) N

      H = (B-A)/N
      M = N/2

      SUM = 0.0
      X = A
      F1 = F(X)
      DO 10 I = 1,M
         F2 = F(X+H)
         F3 = F(X+2*H)
         SUM = SUM + F1 + 4*F2 + F3
         F1 = F3
         X = X + 2*H
10    CONTINUE

      ICS = SUM * H/3.0

      WRITE(*,*)
      WRITE(*,*) INTEGRAL FROM', A, '  TO', B
      WRITE(*,*)
      WRITE(*,*) 'WHEN H = ',H, '  IS', ICS
      WRITE(*,*)

      STOP
      END

---------------- End of main SIMS1 ----------------*
----------------------------------------------------*
Function subprogram F(X)                            *
----------------------------------------------------*

      REAL FUNCTION F(X)
      REAL X

      F = 1-EXP(-X/2.0)

      RETURN
      END

-------------- End of function F(X) ---------------*
```

***Test Run Results*** Output of the program for integrating the function

$$f(x) = 1 - e^{-x/2}$$

from 0.0 to 10.0 is given below:

```
Initial value of X
0.0
Final value of X
10.0
Number of segments (EVEN number)
20

INTEGRAL FROM       .0000000   To  10.0000000

WHEN  H  =  5.000000E-001      IS          8.0134330
Stop - Program terminated.
```

## 12.5 SIMPSON'S 3/8 RULE

Simpson's 1/3 rule was derived using three sampling points that fit a quadratic equation. We can extend this approach to incorporate four sampling points so that the rule can be exact for $f(x)$ of degree 3. Remember, even Simpson's 1/3 rule, although it is based on three points, is third-order accurate. However, a formula based on four points can be used even when the number of segments is odd.

By using the first four terms of Eq. (12.5) and applying the same procedure followed in the previous case, we can show that

$$\boxed{I_{s2} = \frac{3h}{8}\,[f(a) + 3f(x_1) + 3f(x_2) + f(b)]} \qquad (12.16)$$

where $h = (b - a)/3$. This equation is known as *Simpson's 3/8 rule*. This is also known as *Newton's three-eighths rule*.

Similarly, we can show that, using the fifth term of Eq. (12.5), the truncation error of Simpson's 3/8 rule is

$$\boxed{E_{ts2} = -\frac{3h^5}{80} f^{(4)}(\theta_x) = -\frac{(b-a)^5}{6480} f^{(4)}(\theta_x)} \qquad (12.17)$$

where $a < \theta_x < b$.

For a given interval $(a, b)$, the truncation error of Simpson's 1/3 rule is

$$E_{ts1} = -\frac{h^5}{90} f^{(4)}(\theta_x) = -\frac{(b-a)^5}{2880} f^{(4)}(\theta_x)$$

This shows that the 3/8 rule is slightly more accurate than the 1/3 rule.

12.5

**Example [illegible]**

Use Simpson's 3/8 rule to evaluate

$$\text{(a)} \int_1^2 (x^3 + 1)\,dx \qquad \text{(b)} \int_0^{\pi/2} \sqrt{\sin(x)\,dx}$$

*Case (a)*

Basic Simpson's 3/8 rule is based on four sampling points and, therefore, $n = 3$.

$$I_{s2} = \frac{3h}{8}[f(a) + 3f(x_1) + 3f(x_2) + f(b)]$$

$$h = \frac{b-a}{a} = \frac{1}{3}$$

$$x_1 = a + h = 1 + 1/3 = 4/3$$

$$x_2 = a + 2h = 1 + 2/3 = 5/3$$

on substitution of these values, we obtain

$$I_{s2} = \frac{1}{8}[f(1) + f(2) + 3f(4/3) + 3f(5/3)]$$

$$= 4.75$$

Note that the answer is exact. This is expected because Simpson's 3/8 rule is supposed to be exact for cubic polynomials.

*Case (b)*

$$I = \int_0^{\pi/2} \sqrt{\sin(x)}\,dx$$

Here again, $n = 3$ and the integral is given by

$$I_{s2} = \frac{3h}{8}[f(a) + 3f(x_1) + 3f(x_2) + f(b)]$$

$$h = \frac{b-a}{3} = \frac{\pi}{6}$$

$$x_1 = a + h = \frac{\pi}{6}$$

$$x_2 = a + 2h = \frac{\pi}{3}$$

on substitution of these values, we obtain

$$I_{s2} = \frac{\pi}{16}[f(0) + 3f(\pi/6) + f(\pi/3) + f(\pi/2)]$$

$$= \frac{\pi}{16}[0 + 2.12132 + 2.79181 + 1.0]$$

$$= 1.16104$$

## [illegible] HIGHER ORDER RULES

There is no limit to the number of sampling points that could be incorporated in the derivation of Newton-Cotes rule. For instance, we can use a five-point rule to fit exactly the function $f(x)$ of degree 9 and so on. Since the repeated use of lower-order rules provide sufficient accuracy of the estimates, higher-order methods are rarely used.

One more rule which is sometimes used is *Boole's rule* based on five sampling points. This is given by

$$\boxed{I_b = \frac{2h}{45}\,(7f_0 + 32f_1 + 12f_2 + 32f_3 + 7f_3)} \qquad (12.18)$$

where $h = (b - a)/4$

The truncation error of Boole's rule is

$$\boxed{E_{tb} = -\frac{8h^7}{945}\,f^{(6)}\,(\theta_x)} \qquad (12.19)$$

### Example 12.6

Use Boole's five-point formula to compute

$$\int_0^{\pi/2} \sqrt{\sin(x)}\;\mathrm{d}x$$

and compare the results with those obtained in previous examples.

$n = 4,\ h = \pi/8$

$f_0 = 0$

$f_1 = f(\pi/8) \;\;= 0.61861$

$f_2 = f(\pi/4) \;\;= 0.84090$

$f_3 = f(3\pi/8) = 0.96119$

$f_4 = f(\pi/2) \;\;= 1.0$

$$I_b = \frac{\pi}{180}\,[0 + 32(0.61861 + 0.96119) + 12(0.84090) + 7(1.0)]$$

$$= 1.18062$$

The table below shows the results of

$$\int_0^{\pi/2} \sqrt{\sin(x)}\;\mathrm{d}x$$

obtained by various Newton-Cotes rules.

| *Rule* | *n* | *Result* |
|---|---|---|
| Trapezoidal (simple) | 1 | 0.78540 |
| Trapezoidal (composite) | 2 | 1.05314 |
| Simpson's 1/3 (simple) | 2 | 1.14238 |
| Simpson's 3/8 | 3 | 1.16104 |
| Simpson's 1/3 (composite) | 4 | 1.17823 |
| Boole's rule | 4 | 1.18062 |
| Simpson's 1/3 (composite) | 6 | 1.18728 |
| Simpson's 1/3 (composite) | 12 | 1.19429 |

The estimate can be further improved by using still more intervals.

Table 12.1 lists the basic Newton-Cotes rules.

**Table 12.1** Basic Newton-Cotes rules

| *Name* | *Intervals (n)* | *Formula* | *Error* |
|---|---|---|---|
| Trapezoidal | 1 | $\frac{h}{2}(f_0 + f_1)$ | $-\frac{h^2}{12} f''(\theta)$ |
| Simpson's 1/3 | 2 | $\frac{h}{3}(f_0 + 4f_1 + f_2)$ | $-\frac{h^5}{90} f^{(4)}(\theta)$ |
| Simpson's 3/8 | 3 | $\frac{3h}{8}(f_0 + 3f_1 + 3f_2 + f_3)$ | $-\frac{3h^5}{80} f^{(4)}(\theta)$ |
| Boole's rule | 4 | $\frac{2h}{45}(7f_0 + 32f_1 + 12f_2 + 32f_3 + 7f_4)$ | $-\frac{8h^7}{945} f^{(6)}(\theta)$ |

## 12.7 ROMBERG INTEGRATION

It is clear from the discussions we had so far that the accuracy of a numerical integration process can be improved in two ways:

1. By increasing the number of subintervals (i.e. by decreasing $h$)—this decreases the magnitude of error terms. Here, the order of the method is fixed.
2. By using higher-order methods—this eliminates the lower-order error terms. Here, the order of the method is varied and, therefore, this method is known as *variable-order approach*.

The variable-order method can be implemented using Richardson's extrapolation technique discussed in the previous chapter. As we know, this technique involves combining two estimates of a given order to obtain a third estimate of higher order. The method that incorporates this process (i.e. Richardson's extrapolation) to the trapezoidal rule is called *Romberg integration*.

According to the Euler-Maclaurin formula, the error expansion for trapezoidal rule approximation to a definite integral is of the form

$$\int_a^b f(x)\mathrm{d}x - T(h) = a_2h^2 + a_4h^4 + a_6h^6 + \dots \tag{12.20}$$

$T(h)$ is the trapezoidal approximation with step size $= (b - a)/n = h$. Let us define

$$T(h, 0) = T(h)$$

to indicate that $T(h)$ is the trapezoidal rule with no Richardson's extrapolation being applied (zero level extrapolation). Thus, Eq. (12.20) can be written as

$$I = T(h, 0) + a_2h^2 + a_4h^4 + a_6h^6 + \dots \tag{12.21}$$

Let us have another estimate with step size $= (b - a)/2n = h/2$ (at zero level extrapolation) as

$$I = T(h/2, 0) + \frac{a_2}{4}h^2 + \frac{a_4}{16}h^4 + \frac{a_6}{32}h^6 + \dots \tag{12.22}$$

By multiplying Eq. (12.22) by 4 and then subtracting Eq. (12.21) from the resultant equation, we obtain (after rearranging terms),

$$I = \frac{4T(h/2, 0) - T(h, 0)}{4 - 1} + b_4h^4 + b_6h^6 + \dots$$

$$= T(h/2, 1) + b_4h^4 + b_6h^6 + \dots \tag{12.23}$$

where

$$T(h/2, 1) = \frac{4T(h/2, 0) - T(h, 0)}{3}$$

is the *corrected* trapezoidal formula using Richardson's extrapolation technique "once" (level 1). Note that its truncation error is of the order $h^4$, instead of $h^2$ which is the order in the "uncorrected" trapezoidal formula.

Now, we can apply Richardson's extrapolation technique once more to Eq. (12.23) to eliminate the error term containing $h^4$. The result would be

$$I = \frac{16T(h/4, 1) - T(h/2, 1)}{16 - 1} + C_6h^6 + \dots$$

$$= T(h/4, 2) + C_6h^6 + \dots \tag{12.24}$$

where

$$T(h/4, 2) = \frac{16T(h/4, 1) - T(h/2, 1)}{16 - 1}$$

is the estimate, refined again by applying Richardson's extrapolation a second time (level 2). Similarly, we can obtain an estimate with third-level correction as

$$T(h/8, 3) = \frac{64T(h/8, 2) - T(h/4, 2)}{64 - 1}$$

The entire process of repeated use of Richardson's extrapolation technique can be represented in general form as

$$\boxed{T(h/2^i, j) = \frac{4^j T(h/2^i, j-1) - T(h/2^{i-1}, j-1)}{4^j - 1}} \qquad (12.25)$$

where $i = 0, 1, 2 \ldots$ denotes the depth of division and $j \leq i$ denotes the level of improvement.

We can further simplify the notation of Eq. (12.25) by defining

$$R_{ij} = T(h/2^i, j)$$

Thus, we have

$$\boxed{R_{ij} = \frac{4^j R_{i,j-1} - R_{i-1,j-1}}{4^j - 1}} \qquad (12.26)$$

Equation (12.26) is known as *Romberg integration formula.* Note that this equation, when expanded, will form a lower-diagonal matrix. The elements of the matrix **R** are computed row by row in the order indicated in Fig. 12.5. The circled numbers indicate the order of computations and the arrows indicate the dependencies of elements. An element at the head end depends on the element at the tail end.

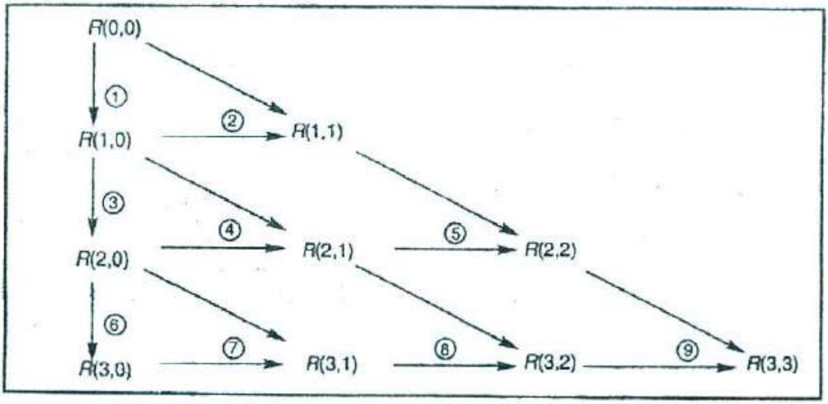


**Fig. 12.5** Order of calculations for Romberg integration

Elements in the first column represent trapezoidal rule at $h$, $h/2$, $h/4$, etc. They can be evaluated recursively as follows:

$$h = b - a$$

$$R(0, 0) = \frac{h}{2} [f(a) + f(b)]$$

$$\boxed{R(i, 0) = \frac{R(i-1, 0)}{2} + h_i \sum_{k=1}^{2^{i-1}} f(x_{2k-1}) \quad \text{for } i = 1, 2, \ldots} \qquad (12.27)$$

where

$$h_i = (b - a)/2^i$$

$$x_k = a + kh_i$$

Equation (12.27) is known as *recursive trapezoidal rule.*

### Example 12.[illegible]

Compute Romberg estimate $R_{22}$ for

$$\int_1^2 1/x \, dx$$

First we apply the basic trapezoidal rule to obtain $R(0,0)$

$$R(0, 0) = \frac{h}{2} [f(a) + f(b)]$$

$$= \frac{2-1}{2}(1 + 1/2) = 0.75$$

Now, we obtain $R(1, 0)$ and $R(2, 0)$ using equation (12.27)

$$R(1, 0) = \frac{R(0,0)}{2} + h_1 f(x_1)$$

$$= \frac{0.75}{2} + \frac{1}{2} \times \frac{1}{1.5} = 0.7083333$$

$$R(2, 0) = \frac{R(1, 0)}{2} + h_2 [f(x_1) + f(x_3)]$$

$$= \frac{0.7083333}{2} + \frac{1}{4} [f(1.25) + f(1.75)]$$

$$= 0.6970237$$

Now, Romberg approximations can be obtained using Eq. (12.26).

$$R(1, 1) = \frac{4R(1, 0) - R(0, 0)}{3}$$

$$= \frac{4(0.7083333) - 0.75}{3} = 0.6944444$$

$$R(2, 1) = \frac{4R(2, 0) - R(1, 0)}{3}$$

$$= \frac{4(0.6970237) - 0.7083333}{3} = 0.6932538$$

$$R(2, 2) = \frac{16R(2, 1) - R(1, 1)}{15}$$

$$= \frac{16(0.6932538) - 0.6944444}{15} = 0.6931744$$

Correct answer = $I_n(2)$ = 0.6931471

Error = 0.0000273

## Program ROMBRG

Computer algorithm for implementing Romberg integration is simple and straight-forward. Starting from the element $R(0, 0)$, all the other elements are calculated row by row. The elements in the first column are calculate using the recursive trapezoidal rule (Eq. (12.27)) and the remaining elements are calculated using the Romberg integration formula (Eq. (12.26)). The process is terminated when two diagonal elements $R(i-1, j-1)$ and $R(i, j)$ agree to the required level of accuracy.

Program ROMBRG implements the steps involved in Romberg integration.

```
* ------------------------------------------------------------ *
      PROGRAM ROMBRG
* ------------------------------------------------------------ *
* Main program                                                  *
*     This program performs Romberg integration                *
*     by bisecting the intervals N times                       *
* ------------------------------------------------------------ *
* Functions invoked                                             *
*     F,ABS                                                     *
* ------------------------------------------------------------ *
* Subroutines used                                              *
*     NIL                                                       *
* ------------------------------------------------------------ *
* Variables used                                                *
*     A - Starting point of the interval                       *
*     B - End point of the interval                            *
*     H - Width of the interval                                *
*     N - Number of times bisection is done                    *
*     M - Number of trapezoids                                 *
*     R - Matrix of Romberg integral values                    *
* ------------------------------------------------------------ *
* Constants used                                                *
*     EPS - Error bound                                         *
* ------------------------------------------------------------ *

      REAL A,B,EPS,H,R,SUM,F,X,ABS
      INTEGER N,M
      INTRINSIC ABS
      EXTERNAL F
      PARAMETER( EPS = 0.00001 )
      DIMENSION R(10,10)
      WRITE(*,*) 'Input endpoints of the interval'
      READ(*,*) A,B
      WRITE(*,*) 'Input maximum number of times'
      WRITE(*,*) 'the subintervals are bisected'
      READ(*,*) N
* Compute using entire interval as one trapezoid
      H = B-A
      R(1,1) = H * (F(A) + F(B))/2.0
      WRITE(*,*)
      WRITE(*,*) R(1,1)
      DO 30 I = 2, N+1
*       Determine number of trapezoids for I_th refinement
      M = 2**(I-2)
*       Reduce step size for I_th refinement
      H = H/2
* Use recursive trapezoidal rule for M strips
      SUM = 0.0
      DO 10 K = 1,M
          X = A+(2*K-1)*H
          SUM = SUM + F(X)
          R(I,1) = R(I-1,1)/2.0+H*SUM
10    CONTINUE
* Compute Richardson's improvements
      DO 20 L = 2,I
          R(I,L)= R(I,L-1)+(R(I,L-I) - R(I-1,L-1))/(4**(L-1)-1)
20    CONTINUE
* Write the results of improvements for I_th refinement
      WRITE(*,*) (R(I,L),L = 1,I)
* Test for desired accuracy
      IF(ABS(R(I-1,I-1) - R(T,I)) .LE. EPS) THEN
*         Stop further refinement
          WRITE(*,*)
          WRITE(*,*) 'ROMBERG INTEGRATION =', R(I,I)
          WRITE(*,*)
          GO TO 40
      ENDIF
* Continue with the refinement process
30 CONTINUE
* write the final result
      WRITE(*,*)
      WRITE(*,*) 'ROMBERG INTEGRATION = ', R(N+1,N+1)
      WRITE(*,*) '(Exit from loop)'
      WRITE(*,*)
```

```
40  STOP
    END
* ----------------End of main ROMBRG --------------- *
* -------------------------------------------------- *
*   Function subprogram F(X)                          *
* -------------------------------------------------- *

    REAL FUNCTION F(X)
    REAL X
    F = 1.0/X
    RETURN
    END
* --------------- End of function F(X) -------------- *
```

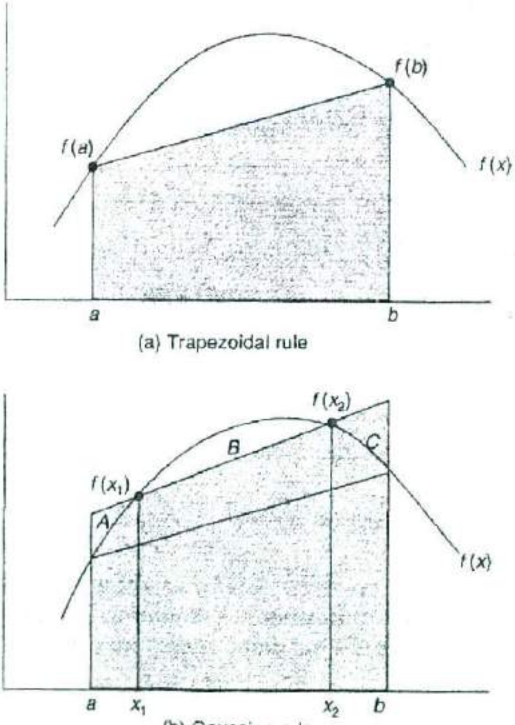
### Test Run Results

```
First run
      Input endpoints of the interval
      1 2
      Input maximum number of times
      the subintervals are bisected.
      1

           7.500000E-001
           7.083334E-001  6.944445E-001

      ROMBERG INTEGRATION = 6.944445E-001
      (Exit from loop)

Second run
      Input endpoints of the interval
      1 2
      Input maximum number of times
      the subintervals are bisected
      2

        7.500000E-001
        7.083334E-001    6.944445E-001
        6.970239E-001    6.932541E-001    6.931747E-001

      ROMBERG INTEGRATION = 6.931747E-001

      (Exit from loop)
```

## 12.8 GAUSSIAN INTEGRATION

We have discussed so far a set of rules based on the Newton-Cotes formula. Recall that the Newton-Cotes formula was derived by integrating the Newton-Gregory forward difference interpolating polynomial. Consequently, all the rules were based on evenly faced sampling points (function values) within the range of integral.

Gauss integration is based on the concept that the accuracy of numerical integration can be improved by choosing the sampling points wisely, rather than on the basis of equal spacing. For example, consider a simple trapezoidal rule as shown in Fig. 12.6(a). Here, the end points of the integral lie on the function curve. Now, consider Fig. 12.6(b). Here, the straight line has been moved up such that area $B = A + C$. Notice that the sampling points are moved away from the end points. The function values at the end points are not used in computation. Rather, function values $f(x_1)$ and $f(x_2)$ are used to compute the shaded area. It is clear that the area obtained from Fig. 12.6(b) would be much closer to the actual area compared to the shaded area in Fig. 12.6(a). The problem is to compute the values of $x_1$ and $x_2$ given the values $a$ and $b$ and to choose appropriate "weights" $w_1$ and $w_2$. The method of implementing the strategy of finding appropriate values of $x_i$ and $w_i$ and obtaining the integral of $f(x)$ is called the *Gaussian integration* or *quadrature*.



(a) Trapezoidal rule

(b) Gaussian rule

**Fig. 12.6** Gaussian integration

Gauss integration assumes an approximation of the form

$$I_g = \int_{-1}^{1} f(x)\,\mathrm{d}x \approx \sum_{i=1}^{n} w_i f(x_i) \tag{12.28}$$

Equation (12.28) contains $2n$ unknowns to be determined. These unknowns can be determined using the condition given in the integration formula (12.28). This should give the exact value of the integral for polynomials of as high a degree as possible.

Let us find the Gaussian quadrature formula for $n = 2$. In this case, we need to find the values of $w_1$, $w_2$, $x_1$ and $x_2$. Let us assume that the integral will be exact up to cubic polynomials. This implies that the functions 1, $x$, $x^2$ and $x^3$ can be numerically integrated to obtain exact results.

$$w_1 + w_2 = \int_{-1}^{1} \mathrm{d}x = 2$$

$$w_1 x_1 + w_2 x_2 = \int_{-1}^{1} x\,\mathrm{d}x = 0$$

$$w_1 x_1^2 + w_2 x_2^2 = \int_{-1}^{1} x^2\,\mathrm{d}x = \frac{2}{3}$$

$$w_1 x_1^3 + w_2 x_2^3 = \int_{-1}^{1} x^3\,\mathrm{d}x = 0$$

Solving these simultaneous equations, we obtain

$$w_1 = w_2 = 1$$

$$x_1 = -\frac{1}{\sqrt{3}} = -0.5113502$$

$$x_2 = \frac{1}{\sqrt{3}} = 0.5773502$$

Thus, we have the Gaussian quadrature formula for $n = 2$ as

$$\int_{-1}^{1} f(x)\,\mathrm{d}x = f\left(-1/\sqrt{3}\right) + f\left(1/\sqrt{3}\right) \tag{12.29}$$

This formula will give correct value for integral of $f(x)$ in the range (–1, 1) for any function up to third-order. Equation (12.29) is also known as *Gauss-Legendre* formula. Two-point Gauss quadrature is illustrated in Fig. 12.7.

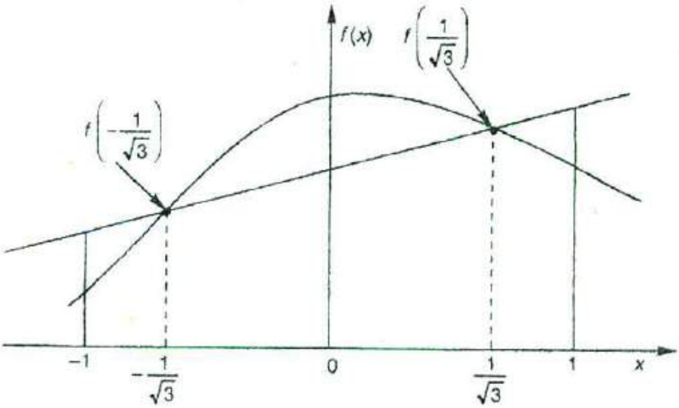

$f(x)$
$f\left(\frac{1}{\sqrt{3}}\right)$
$f\left(-\frac{1}{\sqrt{3}}\right)$
$-1$
$-\frac{1}{\sqrt{3}}$
0
$\frac{1}{\sqrt{3}}$
1
$x$

**Fig. 12.7** Illustration of Gauss-Legendre formula

### Example 12.8

Compute $\int_{-1}^{1} e^{x} \mathrm{d}x$ using two-point Gauss-Legendre formula

$$I = \int_{-1}^{1} \exp(x)\,\mathrm{d}x$$

$$= f(x_1) + f(x_2)$$

where $x_1$ and $x_2$ are Gaussian quadrature points and are given by

$$x_1 = -\frac{1}{\sqrt{3}} = -0.5773502$$

$$x_2 = +\frac{1}{\sqrt{3}} = 0.5773502$$

Therefore,

$$I = \exp(-0.5773502) + \exp(0.5773502)$$
$$= 0.5613839 + 1.7813122$$
$$= 2.3426961$$

## Changing Limits of Integration

Note that the Gaussian formula imposes a restriction on the limits of integration to be from $-1$ to 1. This restriction can be overcome by using the technique of "interval transformation" used in calculus. Let

$$\int_{a}^{b} f(x)\,\mathrm{d}x = C \int_{-1}^{1} g(z)\,\mathrm{d}z$$

Assume the following transformation between $x$ and the new variable $z$.

$$x = Az + B$$

This must satisfy the following conditions:

At

$$x = a, \qquad z = -1 \qquad \text{and } x = b, \qquad z = 1$$

That is

$$B - A = a$$
$$A + B = b$$

Then

$$A = \frac{b-a}{2} \qquad \text{and} \qquad B = \frac{a+b}{2}$$

Therefore

$$x = \frac{b-a}{2} z + \frac{a+b}{2}$$

$$\mathrm{d}x = \frac{b-a}{2} \mathrm{d}z$$

This implies that

$$C = \frac{b-a}{2}$$

Then the integral becomes

$$\frac{b-a}{2} \int_{-1}^{1} g(z)\, \mathrm{d}z$$

The Gaussian formula for this integration is

$$\frac{b-a}{2} \int_{-1}^{1} g(z)\, \mathrm{d}z = \frac{(b-a)}{2} \sum_{i=1}^{n} w_i g(z_i)$$

where $w_i$ and $z_i$ are the weights and quadrature points for the integration domain (–1, 1)

### Example 12.9

Compute the integral

$$I = \int_{-2}^{2} e^{-x/2} \mathrm{d}x$$

using Gaussian two-point formula.

$n$= 2 and therefore

$$I_g = \frac{b-a}{2}\ [w_1 g(z_1) + w_2 g(z_2)]$$

$$x = \frac{b-a}{2} z + \frac{b+a}{2} = 2z$$

Therefore,

$$g(z) = e^{-2z/2} = e^{-z}$$

For a two-point formula

$$w_1 = w_2 = 1$$

$$z_1 = -\frac{1}{\sqrt{3}}$$

$$z_2 = \frac{1}{\sqrt{3}}$$

Upon substitution of these values, we get

$$I_g = 2\ [\exp(-1/\sqrt{3}) + \exp(1/\sqrt{3})]$$
$$= 4.6853922$$

## Higher-Order Gaussian Formulae

By using a procedure similar to the one applied in deriving two-point formula, we can obtain the parameters $w_i$ and $z_i$ for higher-order versions of Gaussian quadrature. These parameters for formulae up to an order of six are tabulated in Table 12.2.

**Table 12.2** Parameters for Gaussian integration

| $n$ | $i$ | $w_i$ | $z_i$ |
|---|---|---|---|
| 2 | 1 | 1.00000 | −0.57735 |
| | 2 | 1.00000 | 0.57735 |
| 3 | 1 | 0.55556 | −0.77460 |
| | 2 | 0.88889 | 0.00000 |
| | 3 | 0.55556 | −0.77460 |
| 4 | 1 | 0.34785 | −0.86114 |
| | 2 | 0.65215 | −0.33998 |
| | 3 | 0.65215 | +0.33998 |
| | 4 | 0.34785 | 0.86114 |
| 5 | 1 | 0.23693 | −0.90618 |
| | 2 | 0.47863 | −0.53847 |
| | 3 | 0.56889 | 0.00000 |
| | 4 | 0.47863 | 0.53847 |
| | 5 | 0.23693 | 0.90618 |

*(Contd.)*

**Table 12.2** *(Contd.)*

| $n$ | $i$ | $w_i$ | $z_i$ |
|---|---|---|---|
| 6 | 1 | 0.17132 | −0.93247 |
| | 2 | 0.36076 | −0.66121 |
| | 3 | 0.46791 | −0.23862 |
| | 4 | 0.46791 | 0.23862 |
| | 5 | 0.36076 | 0.66121 |
| | 6 | 0.17132 | 0.93247 |

### Example 12.[illegible]

Use Gauss-Legendre three-point formula to evaluate

$$\int_2^4 (x^4 + 1)\,\mathrm{d}x$$

Given $n = 3$, $a = 2$, and $b = 2$. Hence

$$I_g = \frac{b-a}{2}\sum_{i=1}^{3} w_i g(z_i)$$

$$= w_1 g(z_1) + w_2 g(z_2) + w_3 g(z_3)$$

$$x = \frac{(b-a)}{2} z + \frac{b+a}{2} = z + 3$$

Therefore,

$$g(z) = (z + 3)^4 + 1$$

For $n = 3$, we have

$$w_1 = 0.55556 \qquad z_1 = -0.77460$$

$$w_2 = 0.88889 \qquad z_2 = 0.0$$

$$w_3 = 0.55556 \qquad z_3 = 0.77460$$

Then

$$\begin{aligned} I_g &= 0.55556\,[(-0.77460 + 3)^4 + 1] \\ &\quad +0.88889\,[(0+3)^4 + 1] \\ &\quad +0.55556[(0.77460 + 3)^4 + 1] \\ &= 14.18140 + 72.88898 + 113.33105 \\ &= 200.40143 \end{aligned}$$

We can verify the answer with analytical solution which is 200.4. Note that three-point Gauss formula should give exact answer for a second order polynomial. The difference in the answer is due to roundoff errors. Roundoff error can be minimised by increasing the precision of Gaussian parameters.

Algorithm 12.1 gives a simple procedure for implementing Gauss-Legendre formula.

**Gaussian Integration**

```
1. Define the function, $f(x)$
2. Obtain integration limits ($a$, $b$)
3. Decide number of interpolating points ($n$)
4. Read the Gaussian parameters ($w_i$, $z_i$)
5. Compute $x_i$ using
       $x_i = \frac{(b-a)}{2} z_i + \frac{b-a}{2}$
6. Compute $I_g$
       $I_g = \frac{b-a}{2} \sum_{i=1}^{n} w_i f(x_i)$
7. Write result
```

**Algorithm 12.1**

## 12.9 SUMMARY

In this chapter, we discussed the integration of definite integrals using numerical integration techniques. The following Newton-Cotes methods were considered in detail:

- Trapezoidal rule
- Simpson's 1/3 rule
- Simpson's 3/8 rule
- Boole's rule

We also presented a method known as Romberg integration to improve the accuracy of the results of the trapezoidal method.

We finally discussed another approach known as Gauss integration which is based on the concept that the accuracy can be improved by choosing the sampling points wisely, rather than equally.

FORTRAN programs were presented for the following methods:

- Trapezoidal rule
- Simpson's 1/3 rule
- Romberg integration

**Key Terms**

*Boole's rule*
*Closed form*
*Newton's three-eighths rule*
*Newton-Cotes formula*

*(Contd.)*

*(Contd.)*

*Composite approach*
*Composite Simpson's 1/3 rule*
*Composite trapezoidal rule*
*Extrapolation*
*Gaussian integration*
*Gaussian quadrature*
*Gauss-Legendre formula*
*Gauss-Legendre rules*
*Integration nodes*
*Lagrange interpolation polynomial*
*Multisegment approach*
*Newton interpolation polynomial*
*Newton-Cotes rules*
*Numerical integration*
*Numerical quadrature*
*Open form*
*Recursive trapezoidal rule*
*Richardson's extrapolation*
*Romberg integration*
*Romberg integration formula*
*Simpson's 1/3 rule*
*Simpson's 3/8 rule*
*Trapezoidal rule*
*Variable-order approach*

## REVIEW QUESTIONS

1. What is numerical integration?
2. When do we need to use a numerical method instead of analytical method for integration?
3. Numerical integration is similar in spirit to the graphical method of finding area under the curve. Explain.
4. Explain the basic principle used in Newton-Cotes methods.
5. Describe the trapezoidal method of computing integrals.
6. What is composite trapezoidal rule? When do we use it?
7. In composite trapezoidal rule, the error is estimated to be inversely proportional to the square of number of segments. That is, we can decrease the error by taking more and more segments. But this does not happen always. Why? Explain.
8. Describe Simpson's method of computing integrals.
9. Prepare a flow chart for implementing the trapezoidal rule.
10. Prepare a flow chart for implementing Simpson's one-third rule.
11. Show that Simpson's one-third rule is exact up to degree 3.
12. Derive Simpson's three-eighth's rule using the first four terms of Newton-Gregory forward formula.
13. State formulae and errcr terms for the four basic Newton-Cotes rules.
14. How could we improve the accuracy of a numerical integration process?
15. What is Romberg integration? How does it improve the accuracy of integration?
16. Explain the concept used in Gaussian quadrature.

## REVIEW EXERCISES

1. Evaluate analytically the following integrals:

   (a) $\int_0^2 (3x^2 + 2x - 5)\, dx$

   (b) $\int_0^2 (3x^3 + 2x^2 - 1)\, dx$

   (c) $\int_0^\pi (3\cos x + 5)\, dx$

2. Evaluate the integrals in Exercise 1 using the basic single segment trapezoidal rule.
3. Evaluate the integrals in Exercise 1 using the basic Simpson's 1/3 rule.
4. Evaluate the integrals in Exercise 1 using the basic Simpson's 3/8 rule.
5. Evaluate the integrals in Exercise 1 using multiple application of the following rules with $n = 4$.
   (a) Trapezoidal rule
   (b) Simpson's 1/3 rule
   (c) Simpson's 3/8 rule
6. Use the trapezoidal rule with $n = 4$ to estimate

$$\int_0^1 \frac{dx}{1+x^2}$$

   correct to five decimal places.

7. Use Simpson's method with $n = 4$ to estimate

$$\int_0^1 \frac{dx}{1+x^2}$$

   correct to five decimal places. Compare this with the result obtained in Exercise 6. How do they compare with the correct answer 0.785398.

8. Estimate the following integrals by (a) trapezoidal method and (b) Simpson's 1/3 method using the given $n$:

   (a) $\int_1^3 \frac{dx}{x}, \qquad n = 2, 4, 8$

   (b) $\int_1^2 \frac{e^x dx}{x}, \qquad n = 4$

(c) $\int_1^5 e^{-x^2}\,dx,$ $\qquad n = 8$

(d) $\int_0^3 \cos^2 x\,dx,$ $\qquad n = 6$

(e) $\int_0^\pi \sqrt{1+3\cos^2 x}\,dx,$ $\qquad n = 6$

(f) $\int_0^2 (e^{x^2} - 1)\,dx,$ $\qquad n = 8$

9. The table below shows the temperature $f(t)$ as a function of time.

| Time, $t$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| Temperature, $f(t)$ | 81 | 75 | 80 | 83 | 78 | 70 | 60 |

(a) Use Simpson's 1/3 method to estimate

$$\int_1^7 f(t)\,dt$$

(b) Use the result in (a) to estimate the average temperature.

10. Use Romberg integration to evaluate

(a) $\int_0^{\pi/2} \frac{\cos x}{\sqrt{1+\sin x}}\,dx$

(b) $\int_0^{3\pi/2} e^x \sin x\,dx$

(c) $\int_1^e \frac{\sqrt{\ln x}}{x}\,dx$

(d) $\int_0^{2\pi} (5 + 2\sin x)\,dx$

(e) $\int_0^2 (e^{x^2} - 1)\,dx$

11. Prove that if $f''(x) > 0$ and $a \le x \le b$, the value of the integral

$$\int_a^b (fxt)\,dx$$

by the trapezoidal rule will always be greater than the exact value of the integral. Verify your conclusion for the following functions:

(a) $f(x) = 2x^3$

(b) $f(x) = 1 + x + x^2$

12. The table below shows the speed of a car at various intervals of time. Find the distance travelled by the car at the end of 2 hours

| Time, hr | 0 | 0.5 | 1.0 | 1.5 | 2.0 | 2.5 |
|---|---|---|---|---|---|---|
| Speed, km/hr | 0 | 40 | 60 | 50 | 45 | 65 |

$f_0$

13. The velocity of a particle is governed by the law

$$v(t) = \frac{\sin(t)}{(t+1)^2 \exp(t)}$$

If the initial position of the particle is $x(0) = 0$, then estimate the position $x(2)$ using the integral

$$x(t) = \int_0^t v(t)\,\mathrm{d}t$$

by applying a suitable Newton-Cotes formula.

14. Estimate the integral

$$I = \int_0^{10} \exp\left(\frac{-1}{1+x^2}\right)\mathrm{d}x$$

by Gauss qudrature, with $n = 2$, 3, and 4.

15. Evaluate the integral

$$I = \int_0^{\pi/2} (1 - 0.25\sin^2 x)^{1/2}\,\mathrm{d}x$$

using Gaussian quadrature. Assume a suitable value of $n$.

16. In an electric circuit, the voltage across the capacitor is given by

$$v(T) = \frac{1}{C}\int_0^T i(t)\,\mathrm{d}t \text{ volts}$$

$$= \frac{10}{C}\int_0^T \sin^2\left(\frac{t}{\pi}\right) \text{volts}$$

Assuming $C = 5F$, compute the value of voltage $v(T)$ for $T = 1$, 2, and 3 seconds.

17. The circumference of an ellipse is given by

$$\text{Circumference} = 4a\int_0^{\frac{\pi}{2}} \sqrt{1 - \left[\frac{(a+b)(a-b)}{a^2}\right]\sin^2\theta\,\mathrm{d}\theta}$$

where $2a$ is the length of major axis and $2b$ is the length of minor axis. Find the circumference if $a = 30$ metres and $b = 20$ metres.

18. The viscous resistance of an object moving through a fluid with velocity $v$ is given by

$$R = -v^{3/2}$$

The velocity is decreasing with time $t$. The time taken for the velocity to decrease from $v_0$ to $v_1$ is given by

$$T = \int_{v_0}^{v_1} \frac{m}{R}\,dv \text{ seconds} \qquad = -\int_{v_0}^{v_1} \frac{m}{v^{3/2}}\,dv \text{ seconds}$$

where $m$ is the mass of the object. Estimate the time $T$ required for an object with $m = 30$ kg to reach a velocity of 10 m/sec from an initial velocity of 20 m/sec.

## PROGRAMMING PROJECTS

1. Write a modular program TRAPE2 that uses the following subprograms as modules to evaluate integrals using the composite trapezoidal rules.
   (a) Input module (Module1)
   (b) Evaluation module (Module2)
   (c) Output module (Module3)

   Use a function subprogram to evaluate the given function.
2. Develop a modular program SIMS2 (similar to TRAPE2) to evaluate integrals using multiple application (i.e. composite) of Simpson's 3/8 rule.
3. Modify the program SIMPS1 to include a test to determine whether $n$ is even and stop the execution if $n$ is odd.
4. Write a program that uses two-point Gaussian quadrature to estimate the integral

$$\int_a^b f(x)\,dx$$

   Split the interval into $n$ equal subintervals and apply the quadrature rule to each subinterval.
5. Show that the integral

$$I_n = \int_0^1 x^2 \; e^{x-1}\,dx, \qquad n = 1, 2, \ldots$$

   can be evaluated recursively by

$$I_n = 1 - n\, I_{n-1} \qquad n = 2, 3, \ldots$$

$$I_1 = \frac{1}{e}$$

Write a program to evaluate $I_{10}$ using this recursive formula anc compare the results by Simpson's rule.

6. Write a program to evaluate the integral of a table of points using Newton's three-eighth's rule.
7. Write a program to experiment with the integration problem

$$I = \int_0^{\pi} \cos(x)\,\mathrm{d}x$$

to see if you can determine an optimum value for $h$ using Simpson's rule.

*Note:* The trapejoidal rule and the Simpson's 1/3 rule can be used to evaluate the integral of a table of points. Development of FORTRAN programs is left as an exercise to the readers. (C programs for evaluating the integral of tabulated data are given in the Appendix D).

CHAPTER 13

# Numerical Solution of Ordinary Differential Equations

## 13.1 NEED AND SCOPE

Many of the laws in physics, chemistry, engineering, biology and economics are based on empirical observations that describe changes in the states of systems. Mathematical models that describe the state of such systems are often expressed in terms of not only certain system parameters but also their derivatives. Such mathematical models, which use differential calculus to express relationship between variables, are known as *differential equations*.

Examples of differential equations are many. A few of them are listed below to illustrate the nature of differential equations that occur in science and engineering.

**1. Law of cooling**

The Newton's law of cooling states that the rate of loss of heat from a liquid is proportional to the difference of temperatures between the liquid and the surroundings. This can be stated in mathematical form as

$$\frac{\mathrm{d}T(t)}{\mathrm{d}t} = k\,(T_s - T(t)) \tag{13.1}$$

where $T_s$ is the temperature of surroundings, $T(t)$ is the temperature of liquid at time $t$ and $k$ is the constant of proportionality.

**2. Law of motion**

The law governing the velocity $v(t)$ of a moving body is given by

$$m\frac{dv(t)}{dt} = F \tag{13.2}$$

where $m$ is the mass of the body and $F$ is the force acting on it.

**3. Kirchhoff's law for an electric circuit**

The voltage across an electric circuit containing an inductance $L$ and a resistance $R$ is given by

$$L\frac{di}{dt} + iR = V \tag{13.3}$$

**4. Radioactive decay**

The radioactive decay of an element is given by

$$\frac{dm}{dt} - km = 0 \tag{13.4}$$

where $m$ is the mass, $t$ is time and $k$ is the constant rate of decay.

**5. Simple harmonic motion**

The equation to describe a simple harmonic motion is given by

$$m\frac{d^2y}{dt^2} + a\frac{dy}{dt} + ky = 0 \tag{13.5}$$

where $y$ denotes displacement and $m$ is the mass. Note that $d^2y/dt^2$ represents acceleration and $dy/dt$ represents velocity of the moving weight.

**6. Force on a moving boat**

When a boat moves through water, the retarding force is proportional to the square of the velocity. The acceleration is given by

$$\frac{dv}{dt} = -\frac{k}{m}v^2 \tag{13.6}$$

where $m$ is the mass and $k$ is the drag coefficient.

**7. Heat flow in a rectangular plate**

The model for heat flow in a rectangle plate that is heated is given by

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{dy^2} = f(x, y) \tag{13.7}$$

where $u(x, y)$ denotes the temperature at point $(x, y)$ and $f(x, y)$ is the heat source.

Note that all these examples contain the rate of change of a variable expressed as a function of variables and parameters. Although most of the differential equations may be solved analytically in their simplest form, analytical techniques fail when the models are modified to take into account the effect of other conditions of real-life situations. In all such cases, numerical approximation of the solution may be considered as a possible approach.

The main concern of this chapter is to present various methods of numerical solution of a class of differential equations known as *ordinary differential equations*. This implies that the differential equations may appear in different forms. This is evident from the examples. We, therefore, define some important classes of differential equations before considering the numerical solution of ordinary differential equations.

## Number of Independent Variables

The quantity being differentiated is called the *dependent variable* and the quantity with respect to which the dependent variable is differentiated is called *independent variable*. If there is only one independent variable, the equation is called an *ordinary differential equation*. If it contains two or more independent variables, the derivatives will be partial and, therefore, the equation is called a *partial differential equation*. Eqs. (13.1) to (13.6) belong to the class of ordinary differential equations while Eq. (13.7) is a partial differential equation. In this chapter we shall consider only the ordinary differential equations.

## Order of Equations ✓ NUB

Differential equations are also classified according to their order. The order of a differential equation is the highest derivative that appears in the equation. When the equation contains only a first derivative, it is called a *first-order differential equation*. For example, Eqs (13.1) to (13.4) are first-order equations. On the other hand, if the highest derivative is a second derivative, the equation is called a *second-order differential equation*. For example, Eq. (13.5) is a second-order equation.

A first-order equation can be expressed in the form

$$\frac{dy}{dx} = f(x, y) \tag{13.8}$$

A second-order equation can be expressed in the form

$$y'' = f(x, y, y') \tag{13.9}$$

where $y''$ denotes the second derivative and $y'$ is the first derivative. Higher-order equations can be reduced to a set of first-order equations by suitable transformations. For example, the equation

$$y'' = f(x, y, y')$$

can be equivalently represented by

$$u' = f(x, y, u)$$

$$y' = u$$

## Degree of Equations ✓ NUB

Sometimes, the equations are referred to by their degree. The degree of a differential equation is the power of the highest-order derivative. For example,

$$xy'' + y^2y' = 2y + 3$$

is a first-degree, second-order equation while,

$$(y''')^2 + 5y' = 0$$

is a second-degree, third-order equation.

## Linear and Nonlinear Equations ✓ NUB

A differential equation is known as a *linear* equation when it does not contain terms involving the products of the dependent variable or its derivatives. For example,

$$y'' + 3y' = 2y + x^2$$

is a second-order, linear equation. The equations

$$y'' + (y')^2 = 1$$
$$y' = -ay^2$$

are nonlinear because the first one contains a product of $y'$ and the second contains a product of $y$.

## General and Particular Solutions

A solution to a differential equation is a relationship between the dependent and independent variables that satisfy the differential equation. For example,

$$y = 3x^2 + x$$

is the solution of

$$y' = 6x + 1$$

Similarly,

$$y = e^x$$

is the solution of

$$y'' = y$$

Note that each of the solutions given above is only one of an infinite number of solutions. For example,

$$y = 3x^2 + x + 2$$
$$y = 3x^2 + x - 10$$

are also solutions of $y' = 6x + 1$. In general, $y' = 6x + 1$ has a solution of the form

$$y' = 3x^2 + x + c$$

where $c$ is known as the constant of integration. Similarly, $y' = y$ has a solution of the form $y = ae^x$. The solution that contains arbitrary constants is not unique and is therefore known as the *general solution*.

If the values of the constants are known, then, on substitution of these values in the general solution, a unique solution known as *particular solution* can be obtained.

## Initial Value Problems

NUB

In order to obtain the values of the integration constants, we need additional information. For example, consider the solution $y = ae^x$ to the equation $y' = y$. If we are given a value of $y$ for some $x$, the constant $a$ can be determined. Suppose $y = 1$ at $x = 0$, then,

$$y(0) = ae^0 = 1$$

Therefore,

$$a = 1$$

and the particular solution is

$$y = e^x$$

If the order of the equation is $n$, we will have to obtain $n$ constants and, therefore, we need $n$ conditions in order to obtain a unique solution. When all the conditions are specified at a particular value of the independent variable $x$, then the problem is called an *initial-value problem*.

It is also possible to specify the conditions at different values of the independent variable. Such problems are called the *boundary-value problems*. For example, if, instead of specifying only $y(0) = 1$, we also specify $y(0) + y(1) = 2$, then the problem will be a boundary-value problem. In this case,

$$y(0) + y(1) = a(1 + e) = 2$$

giving

$$a = 2/(1 + e)$$

## One-step and Multistep Methods

All numerical techniques for solving differential equations involve a series of estimates of $y(x)$ starting from the given conditions. There are two basic approaches that could be used to estimate the values of $y(x)$. They are known as *one-step methods* and *multistep methods*.

In one-step methods, we use information from only one preceding point, i.e. to estimate the value $y_i$, we need the conditions at the previous point $y_{i-1}$ only. Multistep methods use information at two or more previous steps to estimate a value.

## Scope

In this chapter, we mainly concentrate on the solution of ordinary differential equations and discuss the following methods:

1. Taylor series method
2. Euler's method
3. Heun's method

4. Polygon method
5. Runge-Kutta method
6. Milne-Simpson method
7. Adams-Bashforth-Moulton method

The initial-value problem of an ordinary first-order differential equation has the form

$$\boxed{y'(x) = f(x, y(x)),\ y(x_0) = y_0} \tag{13.10}$$

In this chapter, we determine the solution of this equation on a finite interval $(x_0, b)$, starting with the initial point $x_0$. For the sake of simplicity, in a number of places, we use $f$ for $f(x, y)$, $y$ for $y(x)$ and $y^{(k)}$ for $y^{(k)}(x)$.

## 13.2 TAYLOR SERIES METHOD

We can expand a function $y(x)$ about a point $x = x_0$ using Taylor's theorem of expansion

$$y(x) = y(x_0) + (x - x_0)\, y'(x_0) + (x - x_0)^2 \frac{y''(x_0)}{2!}$$

$$+ \ldots + (x - x_0)^n \frac{y^n(x_0)}{n!} \tag{13.11}$$

where $y^i(x_0)$ is the $i$th derivative of $y(x)$, evaluated at $x = x_0$. The value of $y(x)$ can be obtained if we know the values of its derivatives. This implies that if we are given the equation

$$y' = f(x, y) \tag{13.12}$$

we must then repeatedly differentiate $f(x, y)$ implicitly with respect to $x$ and evaluate them at $x_0$.

For example, if $y' = f(x, y)$ then

$$y'' = \frac{d}{dx}\left(\frac{dy}{dx}\right) = \frac{d}{dx}[f(x, y)]$$

$$= \frac{\partial}{\partial x}[f(x, y)] + \frac{\partial}{\partial y}[f(x, y)]\frac{dy}{dx}$$

$$= \frac{\partial f}{\partial x} + \frac{\partial f}{\partial y} f = f_x + f \times f_y \tag{13.13}$$

where $f$ denotes the function $f(x, y)$ and $f_x$ and $f_y$ denote the partial derivatives of the function $f(x, y)$ with respect to $x$ and $y$, respectively. Similarly, we can obtain

$$y''' = f_{xx} + 2f f_{xy} + f^2 f_{yy} + f_x f_y + f f_y^{\,2} \tag{13.14}$$

Let us illustrate this through an example.

Consider the equation

$$y' = x^2 + y^2$$

under the condition $y(x) = 1$ when $x = 0$,

$$y' = x^2 + y^2$$

$$y'' = 2x + 2yy'$$

$$y''' = 2 + 2yy'' + 2(y')^2$$

at $x = 0$, $y(0) = 1$ and, therefore,

$$y'(0) = 1$$

$$y''(0) = 2$$

$$y'''(0) = 2 + (2)(1)(2) + (2)(1)^2 = 8$$

Substituting these values, the Taylor series becomes

$$y(x) = 1 + x + x^2 + \frac{8}{3!}x^3 + \ldots \qquad (13.15)$$

The number of terms to be used depends on the accuracy of the solution needed.

### Example 13.1

NVR3

Use the Taylor method to solve the equation

$$y' = x^2 + y^2$$

for $x = 0.25$ and $x = 0.5$ given $y(0) = 1$

The solution of this equation is given by Eq. (13.15). That is,

$$y(x) = 1 + x + x^2 + 8\frac{x^3}{3!} + \ldots$$

Therefore,

$$y(0.25) = 1 + 0.25 + (0.25)^2 + \frac{8}{6}(0.25)^3 + \ldots$$

$$= 1.33333$$

Similarly,

$$y(0.5) = 1 + 0.5 + 0.5^2 + \frac{8}{6}(0.5)^3 + \ldots$$

$$= 1.81667$$

## Improving Accuracy

The error in Taylor method is in the order of $(x - x_0)^{n+1}$. If $|x - x_0|$ is large, the error can also become large. Therefore, the result of this method in the interval $(x_0, b)$ when $(b - x_0)$ is large, is often found unsatisfactory.

The accuracy can be improved by dividing the entire interval into subintervals $(x_0, x_1)$, $(x_1, x_2)$, $(x_2, x_3)$, ... of equal length and computing $y(x_i)$, $i = 1, 2, ..., n$ successively, using the Taylor series expansion. Here, $y(x_i)$ is used as an initial condition for computing $y(x_{i+1})$. Thus,

$$y(x_{i+1}) = y(x_i) + \frac{y(x_i)}{1!}(x_{i+1} - x_i) + \frac{y''(x_i)}{2!}(x_{i+1} - x_i)^2 + \dots$$

$$\dots + \frac{y^{(m)}(x_i)}{m!}(x_{i+1} - x_i)^m \qquad (13.16)$$

If we denote the size of each subinterval as $h$, then,

$$x_{i+1} - x_i = h \qquad \text{for } i = 0, 1, ..., n - 1$$

and Eq. (13.16) becomes

$$\boxed{y_{i+1} = y_i + \frac{y_i'}{1!}h + \frac{y_i''}{2!}h^2 + \dots + \frac{y_i^{(m)}}{m!}h^m} \qquad (13.17)$$

The derivatives $y_i^{(k)}$ are determined using Eq. (13.12), (13.13) and (13.14) at $x = x_i$ and $y = y_i$. This formula can be used recursively to obtain $y_i$ values.

## Example 13.2

Use the Taylor method recursively to solve the equation

$$y' = x^2 + y^2, \qquad y(0) = 0$$

for the interval (0, 0.4) using two subintervals of size 0.2.

The derivatives of $y$ are given by

$$y' = x^2 + y^2$$

$$y'' = 2x + 2yy'$$

$$y''' = 2 + 2(y')^2 + 2yy''$$

$$y^{(4)} = 6y'\,y'' + 2yy'''$$

$y' = x^2 + y^2$

$y'' = 2x + 2yy'$

$y''' = 2x + 2(y')^2 + 2yy''$

*Iteration 1*

$$y_1 = y_0 + \frac{y_0'}{1!}h + \frac{y_0''}{2!}h^2 + \frac{y_0'''}{3!}h^3 + \frac{y^{(4)}}{4!}h^4 + \dots$$

$$h = 0.2, y_0 = y(0) = 0$$

$$y_0' = y'(0) = 0 + y(0)^2 = 0$$

$$y_0'' = y''(0) = 2 \times 0 + 2 \times y(0) \times y'(0) = 0$$

Similarly,

$$y_0''' = 2$$

$$y_0^{(4)} = 0$$

Therefore,

$$y_1 = 0 + 0 + 0 + \frac{2}{3!}(0.2)^3 + 0$$

$$= 0.002667 \qquad (\text{at } x = 0.2)$$

*Iteration 2*

$$x_1 = 0.2$$

$$y_1 = 0.002667$$

$$y_1' = x_1^2 + y_1^2 = (0.2)^2 + (0.002667)^2 = 0.04$$

$$\begin{aligned} y_1'' &= 2x_1 + 2y_1 y_1' \\ &= 2(0.2) + 2(0.002667)\,(0.04) \\ &= 0.400213 \end{aligned}$$

$$\begin{aligned} y_1''' &= 2 + 2(y_1')^2 + 2y_1 y_1'' \\ &= 2 + 2(0.04)^2 + 2(0,002667)\,(0.400213) \\ &= 2.005335 \end{aligned}$$

$$\begin{aligned} y_1^{(4)} &= 6y_1'\, y_1'' + 2y_1 y_1''' \\ &= 6(0.04)\,(0.400213) + 2(0.002667)\,(2.005335) \\ &= 0.106748 \end{aligned}$$

$$\begin{aligned} y_2 &= y_1 + y_1' h + \frac{y_1''}{2} h^2 + \frac{y_1'''}{6} h^3 + \frac{y^{(4)}}{24} h^4 \\ &= 0.002667 + 0.04(0.2) + \frac{0.400213}{2}(0.2)^2 \\ &\quad + \frac{2.005335}{6}(0.2)^3 + \frac{0.106748}{24}(0.2)^4 \\ &= 0.021352 \end{aligned}$$

That is

$$y(0.4) = 0.021352$$

If we use $h = (b - x_0) = 0.4$ (without subdividing), we obtain

$$y(0.4) = \frac{2}{6}(0.4)^3 = 0.021333$$

---

The correct answer to the accuracy shown is $y(0.4) = 0.021359$. It shows that the accuracy has been improved by using subintervals. The accuracy can be further improved by reducing $h$ further, say, $h = 0.1$.

One major problem with the Taylor series method is the evaluation of higher-order derivatives. They become very complicated. All these derivatives must be evaluated at $(x_i, y_i)$, $i = 0, 1, 2, \ldots$. This method is, therefore, generally impractical from a computational point of view. However, it illustrates the basic approach to numerical solution of differential equations.

## Picard's Method

Consider the differential equation

$$\frac{dy}{dx} = f(x, y)$$

We can integrate this to obtain the solution in the interval $(x_0, x)$

$$\int_{x_0}^{x} dy = \int_{x_0}^{x} f(x, y) dx$$

or

$$y(x) = y(x_0) + \int_{x_0}^{x} f(x, y) dx$$

Since $y$ appears under the integral sign on the right, the integration cannot be formed. The dependent variable $y$ should be replaced by either a constant or a function of $x$. Since we know the initial value of $y$(at $x = x_0$), we may use this as a first appropriation to the solution and the result can be used on the right-hand side to obtain the next appropriation. The iterative equation is written as

$$y^{i+1} = y_0 + \int_{x_0}^{x} f(x, y^{(i)})\ dx \qquad (13.18)$$

Equation (13.18) is known as *Picard's method*. Since this method involves actual integration, sometimes it may not be possible to carry out the integration. Example 13.3 illustrates the application of Picard's method.

It can be seen that Picard's method is not convenient for computer-based solutions. Like Taylor's series method, this is also a semi-numeric method.

### Example 13.3

Solve the following equations by Picard's method

(i) $y'(x) = x^2 + y^2, \quad y(0) = 0$

(ii) $y'(x) = xe^y, \quad y(0) = 0$

and estimate $y(0.1)$, $y(0.2)$ and $y(1)$

(i)

$$y'(x) = x^2 + y^2$$

$$y_0 = 0, \qquad x_0 = 0$$

$$y^{(1)} = y_0 + \int_{x_0}^{x} (x^2 + (y^0)^2)\mathrm{d}x$$

$$= 0 + \int_0^x x^2\,\mathrm{d}x = \frac{x^3}{3}$$

$$y^{(2)} = 0 + \int_{x_0}^{x} (x^2 + (y^1)^2)\mathrm{d}x$$

$$= \int_0^x \left(x^2 + \frac{x^6}{9}\right)\mathrm{d}x = \frac{x^3}{3} + \frac{x^7}{63}$$

This process can be continued further although it may be a difficult task. If we stop at $y^{(2)}$, then

$$y(x) = \frac{x^3}{3} + \frac{x^7}{63}$$

$$y(0.1) = 0.00003333$$

$$y(0.2) = 0.0026667$$

$$y(1) = 0.3492063$$

(ii)

$$y'(x) = xe^y$$

$$y_0 = 0, \qquad x_0 = 0$$

$$y^{(1)} = 0 + \int_0^x xe^0\,\mathrm{d}x = \frac{x^2}{2}$$

$$y^{(2)} = 0 + \int_0^x xe^{(x^2/2)}\,\mathrm{d}x = e^{(x^2/2)} - 1$$

Note that further integrations will become more difficult and even impossible. Now, let us assume

$$y(x) = y^{(2)} = e^{(x^2/2)} - 1$$

$$y(0.1) = 0.0050125$$

$$y(0.2) = 0.0202013$$

$$y(1) = 0.6487213$$

We know that the exact solution of $y'(x) = xe^y$ is

$$y(x) = -\ln\left(1 - \frac{x^2}{2}\right)$$

Therefore

$$y(0.1)_{\text{exact}} = 0.0050125$$
$$y(0.2)_{\text{exact}} = 0.0202027$$
$$y(1)_{\text{exact}} = 0.6933147$$

Note that the error increases when $(x - x_0)$ increases. Better accuracy can be achieved by using the new initial value i.e. $y$ (0.1) can be used as the initial value for computing $y(0.2)$, instead of $y(0)$.

## 13.3 EULER'S ME OD

Euler's method is the simplest one-step method and has a limited application because of its low accuracy. However, it is discussed here as it serves as a starting point for all other advanced methods.

Consider the first two terms of the expansion (13.11)

$$y(x) = y(x_0) + y'(x_0)(x - x_0)$$

Given the differential equation

$$y'(x) = f(x, y) \text{ with } y(x_0) = y_0$$

we have

$$y'(x_0) = f(x_0, y_0)$$

and therefore

$$y(x) = y(x_0) + (x - x_0) f(x_0, y_0)$$

Then, the value of $y(x)$ at $x = x_1$ is given by

$$y(x_1) = y(x_0) + (x_1 - x_0) f(x_0, y_0)$$

Letting $h = x_1 - x_0$, we obtain

$$y_1 = y_0 + h\, f(x_0, y_0)$$

Similarly, $y(x)$ at $x = x_2$ is given by

$$y_2 = y_1 + h\, f(x_1, y_1)$$

In general, we obtain a recursive relation as

$$\boxed{y_{i+1} = y_i + h\, f(x_i, y_i)} \qquad (13.19)$$

This formula is known as *Euler's method* and can be used recursively to evaluate $y_1, y_2, \ldots$ of $y(x_1), y(x_2), \ldots$, starting from the initial condition $y_0 = y(x_0)$. Note that this does not involve any derivatives.

A new value of $y$ is estimated using the previous value of $y$ as the initial condition. Note that the term $h\, f(x_i, y_i)$ represents the incremental value of $y$ and $f(x_i, y_i)$ is the slope of $y(x)$ at $(x_i, y_i)$, i.e. the new value is obtained by extrapolating linearly over the step size $h$ using the slope at its previous value. That is

New value = old value + slope × step size

This is illustrated in Fig. 13.1. Remember that $y_1$ approximates $y(x_1)$ and $y_2$ approximates $y(x_2)$. The difference between them is the error introduced by the method.

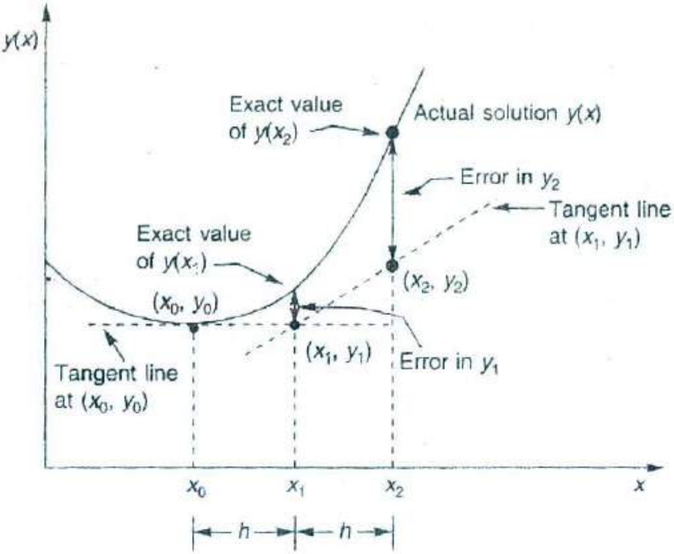


**Fig. 13.1** Illustration of Euler's method for two steps

### Example 13.4

Given the equation

$$\frac{dy}{dx} = 3x^2 + 1 \qquad \text{with } y(1) = 2$$

estimate $y(2)$ by Euler's method using (i) $h = 0.5$ and (ii) $h = 0.25$.

(i) $h = 0.5$

$$y(1) = 2$$
$$y(1.5) = 2 + 0.5[3(1.0)^2 + 1] = 4.0$$
$$y(2.0) = 4.0 + 0.5[3(1.5)^2 + 1] = 7.875$$

(ii) $h = 0.25$

$$y(1) = 2$$
$$y(1.25) = 2 + 0.25[3(1)^2 + 1] = 3.0$$
$$y(1.5) = 3 + 0.25[3(1.25)^2 + 1] = 4.42188$$
$$y(1.75) = 5.42188 + 0.25[3(1.5)^2 + 1] = 7.35938 = 6.3585 ,$$
$$y(2.0) = 7.35938 + 0.25[3(1.75)^2 + 1] = 9.90626 \; 2 \; 8\cdot\text{[illegible]}$$

Notice the difference in answers of $y(2)$ in these two cases. The accuracy is improved considerably when $h$ is reduced to 0.25 (true answer is 10.0).

## Accuracy of Euler's Method

As usual, the accuracy is affected by two sources of error, namely, roundoff error and truncation error. Roundoff error is always present in a computation and this can be minimised by increasing the precision of calculations.

The major cause in the loss of accuracy is truncation error. This arises because of the use of a truncated Taylor series. Since Euler's method uses Taylor series iteratively, the truncation error introduced in an iteration is propagated to the following iterations. This means the total truncation error in any iteration step will consist of two components—the propagated truncation error and the truncation error introduced by the step itself.

The truncation introduced by the step itself is known as the *local truncation error* and the sum of the propagated error and the local error is called the *global truncation error*. Recall the Taylor series expansion used in Euler's method for estimating the values of $y_i$.

$$y_{i+1} = y_i + y_i' h + \frac{y_i''}{2} h^2 + \frac{y_i'''}{3!} h^3 + \ldots$$

Since only the first two terms are used in Euler's formula, the local truncation error is given by

$$E_{t,i+1} = \frac{y_i''}{2} h^2 + \frac{y_i'''}{3!} h^3 + \frac{y_i^{(4)}}{4!} h^4 + \ldots$$

If the step size $h$ is very small, the higher-order terms may be neglected and therefore

$$E_{t,i+1} = \frac{y_i''}{2} h^2$$

The above analysis assumes that the function $y' = f(x, y)$ has continuous derivatives. The local truncation error of Euler's method is of the order $h^2$. If the final estimation requires $n$ steps, the total (global) truncation error at the target point $b$ will be

$$|E_{tg}| = \sum_{i=1}^{n} c_i h^2 = (c_1 + c_2 + \ldots + c_n)h^2 = nch^2$$

where

$$c = (c_1 + c_2 + \ldots + c_n)/n$$

Since

$$n = (b - x_0)/h,$$

$$|E_{tg}| = (b - x_0)ch$$

### Example 13.5

Compute the errors in the estimates of Example (13.4) when $h = 0.5$

Given

$$y' = 3x^2 + 1$$
$$y'' = 6x$$
$$y''' = 6$$

*Step 1*

$$x_0 = 1, \qquad y_0 = 2$$
$$y_1 = y(1.5) = 4.0 \qquad \text{(from example (13.4))}$$

$$E_{t,1} = \frac{y_0''}{2} h^2 + \frac{y_0'''}{6} h^3$$

$$= \frac{6(1)}{2} (0.5)^2 + \frac{6}{6} (0.5)^3 = 0.875$$

*Step 2*

$$x_1 = 1.5, \qquad y_1 = 4.0$$
$$y_2 = y(2.0) = 7.875 \qquad \text{(from example (13.4))}$$

$$E_{t,2} = \frac{6(1.5)}{2} (0.5)^2 + \frac{6}{6} (0.5)^3 = 1.25$$

Exact solution is

$$y(x) = x^3 + x$$

and therefore

$$\text{True } y(1.5) = 4.875$$
$$\text{True } y(2.0) = 10.000$$

The estimated and true values of $y$ and corresponding errors are tabulated below.

| $x$ | *Estimated y* | *True y* | $E_t$ | *Global error* |
|---|---|---|---|---|
| 1.5 | 4.0 | 4.875 | 0.875 | 0.875 |
| 2.0 | 7.875 | 10.000 | 1.250 | 2.125 |

Note that the local and global errors are equal at the first step and they are different in the second step. The difference between them is the propagated truncation error that results from the first step.

Observe that

$$c_1(0.5)^2 = 0.875 \qquad \text{since } c_1 = 3.5$$
$$c_2(0.5)^2 = 1.250 \qquad \text{since } c_2 = 5.0$$
$$c = (c_1 + c_2)/2 = 4.25$$
$$E_{tg} = c(b - x_0)h = 4.25 \times 1 \times 0.5 = 2.125$$

This confirms the error equations for local and global truncation errors discussed in this section.

## Program EULER

The program EULER estimates the solution of the first order differential equation $y' = f(x, y)$ at a given point using Euler's method (Eq. 13.19). The program is simple and self-explanatory. Note the use of an intrinsic function INT in the line

```
N = INT(XP - X)/(H + 0.5)
```

Given initial value of X and the point of solution XP, this statement computes the number of steps required for evaluation using the specified step-size H. The function INT returns the nearest integer of the real value

$$\frac{XP - X}{H}$$

```
* ------------------------------------------------------- *
      PROGRAM EULER
* ------------------------------------------------------- *
* Main program                                            *
*   This program estimates the solution of the first     *
*   order differential equation y' = f(x, y) at a        *
*   given point using Euler's method                      *
* ------------------------------------------------------- *
* Functions invoked                                       *
*      F, INT                                             *
* ------------------------------------------------------- *
* Subroutines used                                        *
*      NIL                                                *
* ------------------------------------------------------- *
* Variables used                                          *
*    X  - Initial value of independent variable          *
*    Y  - Initial value of dependent variable            *
*   XP  - Point of solution                               *
*    H  - Incremental step-size                           *
*    N  - Number of computational steps required         *
*   DY  - Incremental Y in each step                      *
* ------------------------------------------------------- *
* Constants used                                          *
*      NIL                                                *
* ------------------------------------------------------- *

      REAL X,Y,XP,H,DY,F
      INTEGER N,INT
      EXTERNAL F
      INTRINSIC INT
      WRITE(*,*)
      WRITE(*,*)' SOLUTION BY EULERS METHOD'
      WRITE(*,*)
* Read values
      WRITE(*,*)  'Input initial values of x and y'
      READ(*,*) X,Y
      WRITE(*,*)  'Input x at which y is required'
      READ(*,*) XP
      WRITE(*,*)  'Input step-size h'
      READ(*,*) H
* Compute number of steps required
      N = INT((XP-X)/H+0.5)
* Compute Y recursively at each step
      DO 10 I = 1,N
         DY = H*F(X,Y)
         X = X+H
         Y = Y+DY
         WRITE(*,*) I,X,Y
10    CONTINUE
* write the final result
      WRITE(*,*)
      WRITE(*,*)  'Value of Y at X  =',X,' is', Y
      WRITE(*,*)

      STOP
      END
* ------------- End of main EULER -----------------*
* -----------------------------------------------  *
* Function subprogram                               *
* -----------------------------------------------  *

      REAL FUNCTION F(X,Y)
      REAL X,Y

      F = 2.0 * Y/X

      RETURN
      END
* ---------- End of function F(X,Y) ---------------*
```

***Test Run Results***

```
                    SOLUTION BY EULERS METHOD
 Input initial values of x and y
 1.0 2.0
 Input x at which y is required
 2.0
 Input step-size h
 0.25
           1           1.2500000          3.0000000
           2           1.5000000          4.2000000
           3           1.7500000          5.6000000
           4           2.0000000          7.2000000
 Value of Y at X =         2.0000000 is        7.2000000
 Stop - Program terminated.
```

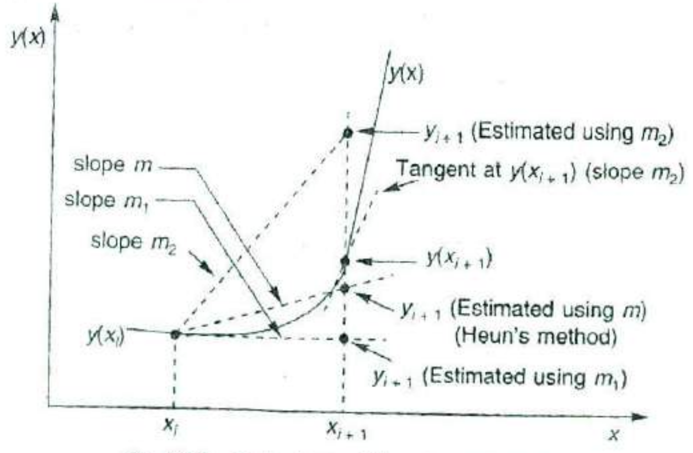
## 13.4 HEUN'S METHOD

Euler's method is the simplest of all one-step methods. It does not require any differentiation and is easy to implement on computers. However, its major weakness is large truncation errors. This is due to its linear characteristic. Recall that Euler's method uses only the first two terms of the Taylor series. In this section, we shall consider an improvement to Euler's method.

In Euler's method, the slope at the beginning of the interval is used to extrapolate $y_i$ to $y_{i+1}$ over the entire interval. Thus,

$$y_{i+1} = y_i + m_1 h$$

where $m_1$ is the slope at $(x_i, y_i)$. As illustrated in Fig. 13.2, $y_{i+1}$ is clearly an underestimate of $y(x_{i+1})$.



$y(x)$
$y(x)$
$y_{i+1}$ (Estimated using $m_2$)
slope $m$
Tangent at $y(x_{i+1})$ (slope $m_2$)
slope $m_1$
slope $m_2$
$y(x_{i+1})$
$y_{i+1}$ (Estimated using $m$)
(Heun's method)
$y(x_i)$
$y_{i+1}$ (Estimated using $m_1$)
$x_i$
$x_{i+1}$
$x$

**Fig. 13.2** Illustration of Heun's method

An alternative is to use the line which is parallel to the tangent at the point $(x_{i+1}, y(x_{i+1}))$ to extrapolate from $y_i$ to $y_{i+1}$ as shown in Fig. 13.2. That is

$$y_{i+1} = y_i + m_2 h$$

where $m_2$ is the slope at $(x_{i+1}, y(x_{i+1}))$. Note that the estimate appears to be overestimated.

A third approach is to use a line whose slope is the average of the slopes at the end points of the interval. Then

$$y_{i+1} = y_i + \frac{m_1 + m_2}{2} h \tag{13.20}$$

As shown in Fig. 13.2, this gives a better approximation to $y_{i+1}$. This approach is known as *Heun's method*.

The formula for implementing Heun's method can be constructed easily. Given the equation

$$y'(x) = f(x, y)$$

we can obtain

$$\begin{aligned} m_1 &= y'(x_i) \qquad = f(x_i, y_i) \\ m_2 &= y'(x_{i+1}) = f(x_{i+1}, y_{i+1}) \end{aligned}$$

and therefore

$$m = \frac{f(x_i, y_i) + f(x_{i+1}, y_{i+1})}{2}$$

Equation (13.20) becomes

$$\boxed{y_{i+1} = y_i + \frac{h}{2} [f(x_i, y_i) + f(x_{i+1}, y_{i+1})]} \tag{13.21}$$

Note that the term $y_{i+1}$ appears on both sides of Eq. (13.21) and, therefore, $y_{i+1}$ cannot be evaluated until the value of $y_{i+1}$ inside the function $f(x_{i+1}, y_{i+1})$ is available. This value can be predicted using the Euler's formula as

$$\boxed{y_{i+1} = y_i + h \times f(x_i, y_i)} \tag{13.22}$$

Then, Heun's formula becomes

$$\boxed{y_{i+1} = y_i + \frac{h}{2}[f(x_i, y_i) + f(x_{i+2}, y^e_{i+1})]} \tag{13.23}$$

Equation (13.23) is an improved version of Euler's method. Since it attempts to correct the values of $y_{i+1}$ using the predicted value of $y_{i+1}$ (by Euler's method), it is classified as a *one-step predictor-corrector* *method*. Eq. (13.22) is known as the *predictor* and Eq. (13.23) is known as the *corrector*. Substituting Eq. (13.22) into Eq. (13.23), we obtain

$$y_{i+1} = y_i + \frac{h}{2}\,[f(x_i, y_i) + f(x_{i+1}, y_i + h\,f(x_i, y_i))] \tag{13.24}$$

### Example 13.6

Given the equation

$$y'(x) = 2y/x \qquad \text{with } y(1) = 2$$

estimate $y(2)$ using (i) Euler's method, and (ii) Heun's method, using $h = 0.25$, and compare the results with exact answers.

Given

$$y' = f(x, y) = 2y/x$$

$$x_0 = 1, \qquad y_0 = 2, \qquad h = 0.25$$

(i) *Euler's method*

$$y(1.25) = y_1 = y_0 + h.\,f(x_0, y_0)$$

$$= 2 + 0.25\,\frac{2 \times 2}{1} = 3.00$$

$$y(1.5) = 3.0 + 0.25\,\frac{2 \times 3.0}{1.25} = 4.2$$

$$y(1.75) = 4.2 + 0.25\,\frac{2 \times 4.2}{1.5} = 5.6$$

$$y(2.0) = 5.6 + 0.25\,\frac{2 \times 5.6}{1.75} = 7.2$$

(ii) *Heun's method*

*Iteration 1*

$$m_1 = \frac{2 \times 2}{1} = 4.0$$

$$y_e(1.25) = 2 + 0.25(4.0) = 3.0$$

$$m_2 = \frac{2 \times 3.0}{1.25} = 4.8$$

$$y(1.25) = 2 + \frac{0.25}{2}\,(4.0 + 4.8) = 3.1$$

*Iteration 2*

$$m_1 = \frac{2 \times 3.1}{1.25} = 4.96$$

$$y_e(1.5) = 3.1 + 0.25(4.96) = 4.34$$

$$m_2 = \frac{2 \times 4.34}{1.5} = 4.8$$

$$y(1.5) = 3.1 + \frac{0.25}{2}\,(4.96 + 5.79) = 4.44$$

*Iteration 3*

$$m_1 = \frac{2 \times 4.44}{1.5} = 5.92$$

$$y_e(1.75) = 4.44 + 0.25(5.92) = 5.92$$

$$m_2 = \frac{2 \times 5.92}{1.75} = 6.77$$

$$y(1.75) = 4.44 + \frac{0.25}{2}\,(5.92 + 6.77) = 6.03$$

*Iteration 4*

$$m_1 = \frac{2 \times 6.03}{1.75} = 6.89$$

$$y_e(2.0) = 6.03 + 0.25(6.89) = 7.75$$

$$m_2 = \frac{2 \times 7.75}{2} = 7.75$$

$$y(2.0) = 6.03 + \frac{0.25}{2}\,(6.89 + 7.75) = 7.86$$

Exact solution of the equation

$$y'(x) = 2y/x \text{ with } y(1) = 2$$

is obtained as

$$y(x) = 2x^2$$

The exact values of $y(x)$ and the estimated values by both the methods are tabulated below.

| $x$ | | $y(x)$ | |
|---|---|---|---|
| | Euler's method | Heun's method | Analytical |
| 1.00 | 2.00 | 2.00 | 2.00 |
| 1.25 | 3.00 | 3.10 | 3.125 |
| 1.50 | 4.20 | 4.44 | 4.50 |
| 1.75 | 5.60 | 6.03 | 6.125 |
| 2.00 | 7.20 | 7.86 | 8.00 |

All estimated values are accurate to two decimal places. It is clear that Heun's method provides better results compared to Euler's method.

## Error Analysis

It can be easily shown that Heun's method is a second-order method and, therefore, its local truncation is of the order $h^3$. Let us consider Eq. (13.24). Letting $i = 0$, for simplicity, we obtain

$$y_1 = y_0 + \frac{h}{2}\,[m_1 + f(x_0 + h,\, y_0 + m_1\, h)] \tag{13.25}$$

Let us expand the term $f(x_{0\,+\,h}, y_0 + m_1 h)$ in a Taylor series form.

$$f(x_0 + h,\, y_0 + m_1 h) = f(x_0, y_0) + h\frac{\partial f}{\partial x} + m_1 h\frac{\partial f}{\partial y}$$

$$= m_1 + hf_x + m_1 hf_y$$

On substituting this into Eq. (13.25) we get

$$y_1 = y_0 + hm_1 + \frac{h^2}{2}(f_x + m_1 f_y) \tag{13.26}$$

We know that

$$m_1 = y'$$

$(f_x + m_1 f_y) = y''$ (see equation (13.13))

and therefore Eq. (13.26) can be written as

$$y_1 = y_0 + \frac{y'}{1!}h + \frac{y''}{2!}h^2$$

This proves that Heun's method is of order $h^2$ and the local truncation error is of the order $h^3$. If the final estimate is obtained after $n$ iterations, then the global truncation error is given by

$$|E_{tg}| = \sum_{i=1}^{n} c_i h^3 = nch^3$$

We know that

$$n = \frac{x_n - x_0}{h} = \frac{b - x_0}{h}$$

Therefore,

$$|E_{tg}| = (b - x_0)ch^2$$

That is, the global truncation error is of the order $h^2$.

## Program HEUN

The algorithm of Heun's method is implemented by the program HEUN. Note that the algorithm is very similar to that of the Euler's method, except for the slope used in extrapolating the initial value.

```
* ---------------------------------------------------------------- *
     PROGRAM HEUN
* ---------------------------------------------------------------- *
* Main program                                                      *
*      This program solves the first order differential             *
*      equation y' = f(x, y) using the Heun's method        *
* --------------------------------------------------------- *
* Functions invoked                                          *
*      F,INT                                                 *
* --------------------------------------------------------- *
* Subroutines used                                           *
*      NIL                                                   *
* --------------------------------------------------------- *
* Variables used                                             *
*      X - Initial value of independent variable             *
*      Y - Initial value of dependent variable               *
*     XP - Point of solution                                 *
*      H - Step-size                                         *
*      N - Number of steps                                   *
* --------------------------------------------------------- *
* Constants used                                             *
*      NIL                                                   *
* --------------------------------------------------------- *

      REAL X,Y,XP,H,M1,M2,F
      INTEGER N,INT
      INTRINSIC INT
      EXTERNAL F

      WRITE(*,*)
      WRITE(*,*)'       SOLUTION BY HEUNS METHOD'
      WRITE(*,*)

* Input values

      WRITE(*,*) 'Input initial values of x and y'
      READ(*,*) X,Y
      WRITE(*,*) 'Input x at which y is required'
      READ(*,*) XP
      WRITE(*,*) 'Input step-size h'
      READ(*,*) H

* Compute number of steps required

      N = INT((XP-X)/H+0.5)

* Compute Y recursively at each step

      DO 20 I = 1,N
         M1 = F(X,Y)
         M2 = F(X+H,Y+M1*H)
         X = X+H
         Y = Y+0.5*H*(M1+M2)
         WRITE(*,*) I,X,Y
```

```
20    CONTINUE
* Write the final result
      WRITE(*,*)
      WRITE(*,*) 'Value of Y at X =', X, '  is', Y
      WRITE(*,*)
      STOP
      END
* ---------------- End of main HEUN ----------------- *
* -------------------------------------------------- *
* Function subprogram                                *
* -------------------------------------------------- *
      REAL FUNCTION F(X,Y)
      REAL X,Y
      F = 2.0 * Y/X
      RETURN
      END
* ------------- End of function F(X,Y) -------------- *
```

### Test Run Results

```
First run
                 SOLUTION BY HEUNS METHOD
      Input initial values of x and y
      1.0 2.0
      Input x at which y is required
      2.0
      Input step-size h
      0.25
                1      1.2500000     3.1000000
                2      1.5000000     4.4433330
                3      1.7500000     6.0302380
                4      2.0000000     7.8608460
      Value of Y at X = 2.0000000 is      7.8608460
      Stop - Program terminated.
Second run
                 SOLUTION BY HEUNS METHOD
      Input initial values of x and y
      1.0 2.0
      Input x at which y is required
      2.0
```

```
    Input step-size h
    0.125
                  1       1.1250000      2.5277780
                  2       1.2500000      3.1175920
                  3       1.3750000      3.7694530
                  4       1.5000000      4.4833640
                  5       1.6250000      5.2593310
                  6       1.7500000      6.0973560
                  7       1.8750000      6.9974420
                  8       2.0000000      7.9595900

   Value of Y at X =   2.0000000 is7.9595900

   Stop - Program terminated.
```

*Third run*

```
                SOLUTION BY HEUNS METHOD

    Input initial values of x and y
    1.0 2.0
    Input x at which y is required
    2.0
    Input step-size h
    0.1
                  1       1.1000000      2.4181820
                  2       1.2000000      2.8761710
                  3       1.3000000      3.3739700
                  4       1.4000000      3.9115800
                  5       1.5000000      4.4890040
                  6       1.6000000      5.1062420
                  7       1.7000000      5.7632950
                  8       1.8000000      6.4601640
                  9       1.9000000      7.1968490
                 10       2.0000000      7.9733510

    Value of Y at X =     2.0000000 is 7.9733510

   Stop - Program terminated.
```

## 13.5 POLYGON METHOD

Another modification of Euler's method is to use the slope of the function at the estimated midpoints of $(x_i, y_i)$ and $(x_{i+1}, y_{i+1})$ to approximate $y_{i+1}$. Thus,

$$y_{i+1} = y_i + f\left(\frac{x_i + x_{i+1}}{2}, \frac{y_i + y_{i+1}}{2}\right)h$$

$$= y_i + f(x_i + h/2, y_i + \Delta y/2)h \qquad (13.27)$$

$\Delta y$ is the estimated incremental value of $y$ from $y_i$ and can be obtained using Euler's formula as

$$\Delta y = h\, f(x_i, y_i)$$

Then, equation (13.27) can be written as

$$\boxed{\begin{aligned} y_{i+1} &= y_i + hf\ (x_i + h/2,\, y_i + h/2\, f(x_i, y_i)) \\ &= y_i + hf(x_i + h/2,\, y_i + m_1 h/2) \\ &= y_i + m_2 h \end{aligned}} \qquad (13.28)$$

where

$$m_1 = f(x_i, y_i) \qquad \text{and} \qquad m_2 = f\left(x_i + \frac{h}{2}, y_i + \frac{m_1 h}{2}\right)$$

Equation (13.28) is known as the *modified Euler's method* or *improved polygon method*. The method, also called the *midpoint method*, is illustrated in Fig. 13.3.

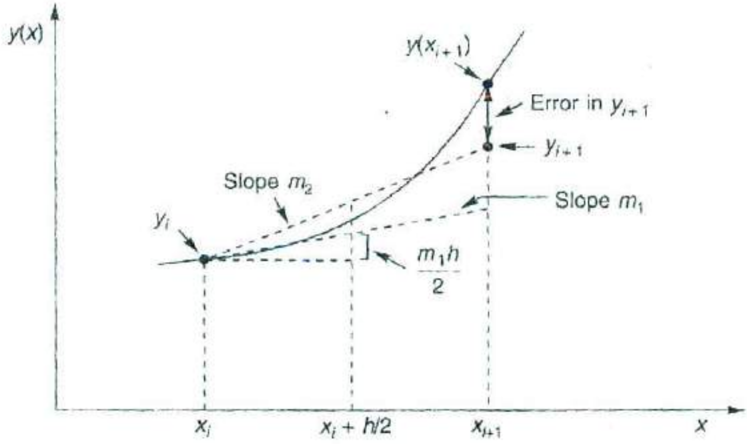


Fig. 13.3 Midpoint method

Like Heun's method, this method is also of the order $h^2$ and therefore, the local truncation error is of the order $h^3$ and the global truncation error is of the order $h^2$.

### Example 13.7

Estimate $y(1.5)$ with $h = 0.25$ for the equation in Example 13.6 using polygon method.

$$y' = f(x, y) = \frac{2y}{x}$$

$$
\begin{aligned}
y(1) &= 2.0 \\
y(1.25) &= 2.0 + 0.25\, f(1 + 0.125,\ 2 + 0.125 f(1, 2)) \\
&= 2.0 + 0.25\, f(1.125,\ 2.5) = 3.11 \\
y(1.5) &= 3.11 + 0.25\, f(1.25 + 0.125,\ 3.11 + 0.125\, f(1.25,\ 3.11) \\
&= 3.11 + 0.25\, f(1.375,\ 3.732) = 4.47
\end{aligned}
$$

Estimated values of $y(1.5)$ by various methods for the equation

$$y\,'(x) = 2y/x \quad \text{with } y(1) = 2.0$$

are given below:

| | | |
|---|---|---|
| Euler's method | : | 4.20 |
| Heun's method | : | 4.44 |
| Polygon method | : | 4.47 |
| Exact answer | : | 4.50 |

Note that the polygon method yields better results compared to the Heun's method.

## Program POLYGN

Program POLYGN implements the polygon algorithm to solve a differential equation of type $y\,' = f(x, y)$

```
* ------------------------------------------------------ *
      PROGRAM POLYGN
* ------------------------------------------------------ *
* Main program                                           *
*     This program solves the differential equation      *
*     of type y' = f(x, y) by polygon method             *
* ------------------------------------------------------ *
* Functions invoked                                      *
*     F,INT                                              *
* ------------------------------------------------------ *
* Subroutines used                                       *
*     NIL                                                *
* ------------------------------------------------------ *
* Variables used                                         *
*     X - Initial value of the independent variable      *
*     Y - Initial value of the dependent variable        *
*    XP - Point of solution                              *
*     H - Incremental step-size                          *
*     N - Number of computational steps required         *
* ------------------------------------------------------ *
* Constants used                                         *
*     NIL                                                *
* ------------------------------------------------------ *
```

```
      REAL X,Y,XP,H,M1,M2,F
      INTEGER N,INT
      INTRINSIC INT
      EXTERNAL F

      WRITE(*,*)
      WRITE(*,*)'          SOLUTION BY POLYGON METHOD'
      WRITE(*,*)

* Input values

      WRITE(*,*) 'Input initial values of x and y'
      READ(*,*) X,Y
      WRITE(*,*) 'Input x at which y is required'
      READ(*,*) XP
      WRITE(*,*) 'Input step-size h'
      READ(*,*) H

* Compute number of steps required

      N = INT((XP-X)/H+0.5)

* Compute Y at each step

      DO 30 I = 1,N
         M1 = F(X,Y)
         M2 = F(X+0.5*H,Y+0.5*H*M1)
         X = X+H
         Y = Y+M2*H
         WRITE(*,*) I,X,Y
30    CONTINUE

* Write the final value of Y

      WRITE(*,*)
      WRITE(*,*) 'Value of Y at X =', X, '  is', Y
      WRITE(*,*)

      STOP
      END

* --------------- End of main POLYGN -----------------*
* ----------------------------------------------------*
* Function subprogram                                 *
* ----------------------------------------------------*

      REAL FUNCTION F(X,Y)
      REAL X,Y

      F = 2.0 * Y/X

      RETURN
      END
* --------------- End of function F(X,Y) --------------*
```

***Test Run Results***

```
              SOLUTION BY POLYGON METHOD
Input initial values of x and y
1.0 2.0
Input x at which y is required
2.0
Input step-size h
0.25
              1        1.2500000       3.1111110
              2        1.5000000       4.4686870
              3        1.7500000       6.0728310
              4        2.0000000       7.9235990

Value of Y at X =   2.0000000 is   7.9235990

Stop - Program terminated.
```

## 13.6 RUNGE-KUTTA METHODS

Runge-Kutta methods refer to a family of one-step methods used for numerical solution of initial value problems. They are all based on the general form of the extrapolation equation,

$$\begin{aligned} y_{i+1} &= y_i + \text{slope} \times \text{interval size} \\ &= y_i + mh \end{aligned}$$

where $m$ represents the slope that is weighted averages of the slopes at various points in the interval $h$. If we estimate $m$ using slopes at $r$ points in the interval $(x_i, x_{i+1})$, then $m$ can be written as

$$m = w_1 m_1 + w_2 m_2 + \ldots + w_r\, m_r \tag{13.29}$$

where $w_1, w_2, \ldots, w_r$ are weights of the slopes at various points. The slopes $m_1, m_2, \ldots, m_r$ are computed as follows:

$$\begin{aligned} m_1 &= f(x_i, y_i) \\ m_2 &= f(x_i + a_1 h, y_i + b_{11}\, m_1 h) \\ m_3 &= f(x_i + a_2 h, y_i + b_{21}\, m_1 h + b_{22}\, m_2 h) \\ &\;\;\vdots \\ m_r &= f(x_i + a_{r-1} h, y_i + b_{r-1,\,1}\, m_1 h + \ldots + b_{r-1,\,r-1}\, m_{r-1}\, h) \end{aligned}$$

That is

$$m_1 = f(x_i, y_i) \qquad r = 1$$

$$m_r = f\left(x_i + a_{r-1} h, y_i + h \sum_{j=1}^{r-1} b_{r-1,j}\, m_j\right) \quad r \geq 2 \tag{13.30}$$

Note that the computation of slope at any point involves the slopes at all previous points. Slopes can be computed recursively using equation (13.30) starting from $m_1 = f(x_i, y_i)$.

Runge-Kutta (RK) methods are known by their order. For instance, an RK method is called the *r-order Runge-Kutta method* when slopes at $r$ points are used to compute the weighted average slope $m$. [illegible] that [illegible] method [illegible] slope at $(x_i, y_i)$ to [illegible], therefore, Euler's method is a *first-order Runge-Kutta method*. Similarly, Heun's method is a *second-order Runge-Kutta method* because it employs slopes at two end points of the interval.

As it was demonstrated by the Euler and Heun methods, higher the order, better would be the accuracy of estimates. Therefore, selection of order for using RK methods depends on the problem under consideration.

## Determination of Weights

For using a Runge-Kutta method, the first requirement is the determination of weights of slopes at various points. The number of points is equal to the order of the method chosen. For the purpose of demonstration, we consider here the second-order Runge-Kutta method and show how to evaluate various constants and weights.

The second-order RK method has the form

$$y_{i+1} = y_i + (w_1 m_1 + w_2 m_2)h \tag{13.31}$$

where

$$m_1 = f(x_i, y_i)$$
$$m_2 = f(x_i + a_1 h, y_i + b_{11}\, m_1 h) \tag{13.32}$$

The weights $w_1$ and $w_2$ and the constants $a_1$ and $b_{11}$ are to be determined. The centre principle of the Runge-Kutta approach is that these parameters are chosen such that a power series expansion of the right side of Eq. (13.31) agrees with the Taylor series of expansion of $y_{i+1}$ in terms of $y_i$ and $f(x_i, y_i)$.

The second-order Taylor series expansion of $y_{i+1}$ about $y_i$ is

$$y_{i+1} = y_i + y'h + \frac{y''}{2!}\, h^2 \tag{13.33}$$

Given

$$y_i' = f(x_i, y_i) = f$$

$$y_i'' = \frac{\mathrm{d}f}{\mathrm{d}x} = f_x + f_y\, f$$

Therefore, Eq. (13.33) becomes

$$y_{i+1} = y_i + fh + (f_x + f_y\, f)\, \frac{h^2}{2} \tag{13.34}$$

Now, consider the right side of Eq. (13.31). Since $m_1$ is already a function of $x_i$ and $y_i$, we need to expand only $m_2$ as a power series in terms of $f(x_i, y_i)$. From Eq. (13.32)

$$m_2 = f(x_i + a_1 h, y_i + b_{11} m_1 h)$$

Expanding the function on the right-hand side using the Taylor series expansion, we get

$$m_2 = f(x_i, y_i) + a_1 h\, f_x + b_{11} m_1 h\, f_y + O(h^2)$$

Substituting this in Eq. (13.31) and replacing $m_1 = f(x_i, y_i)$ by $f$, we get

$$\begin{aligned} y_{i+1} &= y_i + [w_1 f + w_2 f + w_2 a_1 h f_x + w_2 b_{11} h\, f f_y]h + O(h^3) \\ &= y_i + (w_1 + w_2)hf + (w_2 a_1 f_x + w_2 b_{11} f f_y)\, h^2 + O(h^3) \qquad (13.35) \end{aligned}$$

If Eqs (13.34) and (13.35) are to be equivalent, then they should agree term by term. This is possible only if

$$\begin{aligned} w_1 + w_2 &= 1 \\ w_2 a_1 &= 1/2 \\ w_2\, b_{11} &= 1/2 \end{aligned}$$

Note that we have four unknowns and only three equations. Therefore, there is no unique solution. However, we can assume a value for one of the constants and determine the others. This implies that there is an infinite family of second-order RK methods. For example, if we choose $w_1 = 1/2$, then we get

$$w_1 = 1/2, \qquad w_2 = 1/2, \qquad a_1 = 1, \qquad b_{11} = 1$$

With these values, Eq. (13.31) becomes

$$\boxed{y_{i+1} = y_i + \frac{m_1 + m_2}{2} h} \qquad (13.36)$$

where,

$$\begin{aligned} m_1 &= f(x_i, y_i) \\ m_2 &= f(x_i + h, y_i + m_1 h) \end{aligned}$$

Note that this equation is the Heun's formula.

Similarly, if we choose $w_1 = 0$, then we get

$$w_1 = 0, \qquad w_2 = 1, \qquad a_1 = 1/2, \qquad b_{11} = 1/2$$

and Eq. (13.31) becomes

$$\boxed{y_{i+1} = y_i + m_2 h} \qquad (13.37)$$

where

$$m_1 = f(x_i, y_i)$$

$$m_2 = f\left(x_i + \frac{h}{2}, y_i + \frac{m_1 h}{2}\right)$$

This results in the midpoint or polygon method.

Another strategy is to choose the parameters such that the bound on the truncation error is minimum. It has been shown by Ralston that $w_1 = 1/3$ and $w_2 = 2/3$ produces minimum truncation error. With these weights,

$$\boxed{y_{i+1} = y_i + \frac{m_1 + 2m_2}{3} h} \tag{13.38}$$

where,

$$m_1 = f(x_i, y_i)$$

$$m_2 = f\left(x_i + \frac{3}{4}h, y_i + \frac{3}{2}m_1 h\right)$$

## Fourth-Order Runge-Kutta Methods

It is clear from Eq. (13.29) and the discussions above that it is possible to construct RK methods of different orders. However, the commonly used ones are the fourth-order methods. Although there are different versions of fourth-order RK methods, the most popular method is the *classical fourth-order Runge-Kutta* method given below:

$$\boxed{\begin{aligned} m_1 &= f(x_i, y_i) \\ m_2 &= f\left(x_i + \frac{h}{2}, y_i + \frac{m_1 h}{2}\right) \\ m_3 &= f\left(x_i + \frac{h}{2}, y_i + \frac{m_2 h}{2}\right) \\ m_4 &= f(x_i + h, y_i + m_3 h) \\ y_{i+1} &= y_i + \left(\frac{m_1 + 2m_2 + 2m_3 + m_4}{6}\right)h \end{aligned}} \tag{13.39}$$

### Example 13.8

Use the classical RK method to estimate $y(0.4)$ when

$$y'(x) = x^2 + y^2 \quad \text{with} \quad y(0) = 0$$

Assume $h = 0.2$

$$f(x, y) = x^2 + y^2$$

$$m_1 = f(x_0, y_0) = 0$$

$0^2 + 0^2 = 0$

$$m_2 = f\left(x_0 + \frac{h}{2}, y_0 + \frac{m_1 h}{2}\right) = f(0.1, 0) = 0.01$$

$$m_3 = f\left(x_0 + \frac{h}{2}, y_0 + \frac{m_2 h}{2}\right) = f\left(\frac{0.2}{2}, \frac{0.01 \times 0.2}{2}\right) = 0.01$$

$$m_4 = f(x_0 + h, y_0 + m_3 h) = f(0.2, 0.01 \times 0.2) = 0.04$$

$$y(0.2) = 0 + \frac{0 + 2 \times 0.01 + 2 \times 0.01 + 0.04}{6}\, 0.2 = 0.002667$$

*Iteration 2*

$$x_1 = 0.2$$

$$y_1 = 0.002667$$

$$m_1 = f(0.2, 0.002667) = 0.04$$

$$m_2 = f\left(0.3, 0.002667 + \frac{0.04 \times 0.2}{2}\right) = 0.090044$$

$$m_3 = f\left(0.3, 0.002667 + \frac{0.090044 \times 0.2}{2}\right) = 0.090136$$

$$m_4 = f(0.4, 0.002667 + (0.090136)(0.2)) = 0.160428$$

$$y(0.4) = 0.002667 + \frac{0.04 + 2(0.090044) + 2(0.090136) + 0.160428}{6}\, 0.2$$

$$= 0.021360 \text{ (correct to six decimals)}$$

The exact answer is 0.021359. If we use $h = 0.1$, then $y(0.4)$ will be 0.021359. Try and check.

## Program RUNGE4

RUNGE4 is a program designed to compute the solution of a first order differential equation of type $y' = f(x, y)$ using the fourth-order Runge-Kutta method.

```
* -------------------------------------------------------- *
     PROGRAM RUNGE4
* -------------------------------------------------------- *
* Main program                                              *
*   This program computes the solution of first order       *
*   differential equation of type y' = f(x,y) using         *
*   the 4th order Runge-Kutta method                        *
* -------------------------------------------------------- *
* Functions invoked                                         *
*        F,INT                                              *
* -------------------------------------------------------- *
* Subroutines used                                          *
*     NIL                                                   *
* --------------------------------------------------------- *
* Variables used                                            *
*     X - Initial value of independent variable             *
*     Y - Initial value of dependent variable               *
*    XP - Point of solution                                 *
*     H - Step-size                                         *
*     N - Number of steps                                   *
* --------------------------------------------------------- *
* Constants used                                            *
*     NIL                                                   *
* --------------------------------------------------------- *

      REAL X,Y,XP,H,M1,M2,M3,M4,F
      INTEGER N,INT
      INTRINSIC INT
      EXTERNAL F

      WRITE(*,*)
      WRITE(*,*)'     SOLUTION BY 4_TH ORDER R-K METHOD'
      WRITE(*,*)

* Input Values

      WRITE(*,*) 'Input initial values of x and y'
      READ(*,*) X,Y
      WRITE(*,*) 'Input x at which y is required'
      READ(*,*) XP
      WRITE(*,*) 'Input step-size h'
      READ(*,*) H

* Compute number of steps required

      N = INT((XP-X)/H+0.5)

* Compute Y at each step

      WRITE(*,*)'     ------------------------------'
      WRITE(*,*)'     STEP        X           Y   '
      WRITE(*,*)'     ------------------------------'

      DO 40 I = 1,N
         M1 = F(X,Y)
         M2 = F(X+0.5*H,Y+0.5*M1*H)
         M3 = F(X+0.5*H,Y+0.5*M2*H)
         M4 = F(X+H,Y+M3*H)
         X = X+H
         Y = Y+(M1+2.0*M2+2.0*M3+M4)*H/6.0
         WRITE(*,*) I,X,Y
```

```
40    CONTINUE
      WRITE(*,*)'    ----------------------------------'
* Write the final value of Y
      WRITE(*,*)
      WRITE(*,*) 'Value of Y at X =', X,'  is', Y
      WRITE(*,*)
      STOP
      END
* --------------- End of main RUNGE4 ----------------*
* ---------------------------------------------------*
* Function subprogram                                 *
* ---------------------------------------------------*
      REAL FUNCTION F(X,Y)
      REAL X,Y
      F = 2.0 * Y/X
      RETURN
      END
* -------------- End of function F(X,Y) ------------*
```

***Test Run Results***

```
          SOLUTION BY 4_TH ORDER R-K METHOD
      Input initial values of x and y
      1.0 2.0
      Input x at which y is required
      2.0
      Input step-size h
      0.25
            ----------------------------------------------
               STEP          X                Y
            ----------------------------------------------
                 1       1.2500000        3.1246910
                 2       1.5000000        4.4993830
                 3       1.7500000        6.1240550
                 4       2.0000000        7.9986960
            ----------------------------------------------
      Value of Y at X = 2.000000 is 7.9986960
      Stop - Program terminated.
```

## ACCURACY OF ONE-STEP METHODS

How do we achieve the desired level of accuracy in one-step methods? One approach is to repeat the computations at decreasing values of $h$ until the required accuracy is obtained. This may involve a large number of repetitions if the initial value of $h$ is far away from the optimal value. Another approach is to estimate the value of $h$ that is likely to give the desired level of accuracy.

It is very difficult to have a formula in terms $h$ for the global error. However, we know that the order of global truncation error is $h^r$ if the order of local truncation error is $h^{r+1}$. We can use this information to study the sensitivity of the solution to the value of $h$ and thereby estimate the size of $h$. Obtain estimates of $y(h)$ at two different values of $h$, say $h_1$ and $h_2$. Then

$$y_{\text{exact}} - y(b, h_1) = ch_1^r$$

$$y_{\text{exact}} - y(b, h_2) = ch_2^r$$

where $c$ is the constant of proportionality. The above equations can be solved for $c$ as

$$c = \left|\frac{y(b, h_2) - y(b, h_1)}{h_1^r - h_2^r}\right| \tag{13.40}$$

If we need the answer to an accuracy of $d$ decimal places, then the error must not be greater than $0.5 \times 10^{-d}$ and, therefore,

$$ch^r \leq 0.5 \times 10^{-d}$$

Thus

$$h_{\text{opt}}^r = \left|\frac{(h_1^r - h_2^r)\,10^{-d}}{2\Delta y}\right| \tag{13.41}$$

where $\Delta y = y(b, h_2) - y(b, h_1)$. For example, for a second-order method, the global error is proportional to $h^2$ and therefore $h_{\text{opt}}$ is given by

$$h_{\text{opt}} = \sqrt{\left|\frac{(h_1^2 - h_2^2)\,10^{-d}}{2\Delta y}\right|}$$

If we choose $h_1 = 2h_2$, then

$$h_{\text{opt}} = h_1 \sqrt{\left|\frac{3 \times 10^{-d}}{8\Delta y}\right|}$$

Any value of $h < h_{\text{opt}}$ will give the desired accuracy.

## Example 13.9

Two estimates of $y(0.8)$ of the equation

$$y'(x) = x^2 + y^2 \qquad \text{with} \qquad y(0) = 1$$

are obtained using fourth-order RK method at $h = 0.05$ and $h = 0.025$.

$$y(0.8,\ 0.05) = 5.8410870$$

$$y(0.8,\ 0.025) = 5.8479637$$

Estimate the value of $h$ required to obtain the solution accurate to

(i) four decimal places and (ii) six decimal places

(i) $d = 4$

$$h_4 = \frac{(0.05^4 - 0.025^4)\,10^{-4}}{2(0.0068767)} = 4.26 \times 10^{-8}$$

$$h = 0.01437$$

(ii) $d = 6$

$$h^4 = \frac{(0.05^4 - 0.025^4)\,10^{-6}}{2(0.0068767)} = 4.26 \times 10^{-10}$$

$$h = 0.00454$$

We may use $h = 0.01$ to obtain a figure accurate to four decimal places and $h = 0.004$ to obtain a figure accurate to six decimal places.

## 13.8 MULTISTEP METHODS

So far we have discussed many methods for obtaining numerical solution of first-order initial-value problems. All of them use information only from the last computed point $(x_i, y_i)$ to compute the next point $(x_{i+1}, y_{i+1})$. Therefore, all these methods are called *single-step methods*. They do not make use of the information available at the earlier steps, $y_{i-1}$, $y_{i-2}$, etc., even when they are available. It is possible to improve the efficiency of estimation by using the information at several previous points. Methods that use information from more than one previous points to compute the next point are called *multistep methods*. Sometimes, a pair of multistep methods are used in conjunction with each other, one for predicting the value of $y_{i+1}$ and the other for correcting the predicted value of $y_{i+1}$. Such methods are termed *predictor-corrector methods*.

One major problem with multistep methods is that they are not self-starting. They need more information than the initial value condition. If a method uses four previous points, say $y_0$, $y_1$, $y_2$, and $y_3$, then all these values must be obtained before the method is actually used. These values, known as *starting values*, can be obtained using any of the single-step methods discussed earlier. *It is important to note that the degree of accuracy of the single-step method must match that of the multistep method to be used.* For instance, a fourth-order RK method is normally used to generate starting values for implementing a fourth-order multistep method. In this section, we consider the following popular multistep methods:

1. Milne-Simpson method
2. Adams-Bashforth-Moulton method

Both of them are fourth-order methods and use a pair of multistep methods in conjunction with each other.

## Milne-Simpson Method

The Milne-Simpson method is a predictor-corrector method. It uses a Milne formula as a *predictor* and the popular Simpson's formula as a *corrector*. These formulae are based on the fundamental theorem of calculus.

$$y(x_{i+1}) = y(x_j) + \int_{x_j}^{x_{i+1}} f(x, y)\,dx \tag{13.42}$$

When $j = i - 3$, the Eq. becomes an open integration formula and produces the *Milne's formula*

$$\boxed{y_{i+1} = y_{i-3} + \frac{4h}{3}(2f_{i-2} - f_{i-1} + 2f_i)} \tag{13.43}$$

Similarly, when $j = i - 1$, Eq. (13.42) becomes a closed form integration and produces the two-segment *Simpson's formula*

$$\boxed{y_{i+1} = y_{i+1} + \frac{h}{3}(f_{i-1} + 4f_i + f_{i+1})} \tag{13.44}$$

Milne's formula is used to 'predict' the value of $y_{i+1}$ which is then used to calculate $f_{i+1}$ (in Eq. (13.44)) from the differential equation.

$$f_{i+1} = f(x_{i+1}, y_{i+1})$$

Then, Eq. (13.44) is used to correct the predicted value of $y_{i+1}$. The process is then repeated for the next value of $i$. Each stage involves four basic calculations, namely,

1. prediction of $y_{i+1}$
2. evaluation of $f_{i+1}$
3. correction of $y_{i+1}$
4. improved value of $f_{i+1}$ (for use in next stage)

It is also possible to use the corrector formula repeatedly to refine the estimate of $y_{i+1}$ before moving on to the next stage (see Fig. 13.4).

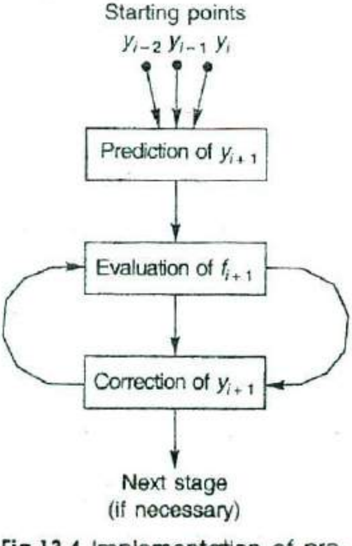


**Fig.13.4** Implementation of predictor-corrector methods

### [illegible]

Given the equation

$$y'(x) = \frac{2y}{x} \qquad \text{with } y(1) = 2$$

estimate $y(2)$ using the Milne-Simpson predictor-corrector method. Assume $h = 0.25$.

Milne's formula at $i = 3$ is

$$y_4^p = y_0 + \frac{4h}{3}(2f_1 - f_2 + 2f_3)$$

Simpson's formula at $i = 3$ is

$$y_4^c = y_2 + \frac{h}{3}\left(f_2 + 4f_3 + f_4^p\right)$$

where $f_i = f(x_i, y_i)$

To use these formulae, we require the estimates of $y_1$, $y_2$ and $y_3$ in addition to the initial condition $y_0$. These can be obtained using any of the single-step fourth-order methods.

Let us assume that they have been estimated using the fourth-order RK method as follows:

$$y_1 = y(1.25) = 3.13$$
$$y_2 = y(1.5\ ) = 4.50$$
$$y_3 = y(1.75) = 6.13$$

Then

$$f_1 = \frac{2 \times 3.13}{1.25} = 5.01$$

$$f_2 = \frac{2 \times 4.5}{1.5} = 6.00$$

$$f_3 = \frac{2 \times 6.13}{1.75} = 7.01$$

Substituting these in Milne's formula we predict the value of $y(2)$ as

$$y_4^p = 2.00 + \frac{4 \times 0.25}{3}\ (2 \times 5.01 - 6.00 + 2 \times 7.01) = 8.01$$

$$f_4^p = \frac{2 \times 8.01}{2} = 8.01$$

Now we obtain the corrected value of $y(2)$ using Simpson formula as

$$y_4^c = 4.50 + \frac{0.25}{3}\ (6.00 + 4 \times 7.01 + 8.01)$$

$$= 8.00$$

We can again use the corrector formula to refine the estimate

$$f_4 = \frac{2 \times 8.00}{2} = 8.00$$

$$y_4^c = 4.50 + \frac{0.25}{3}(6.00 + 4 \times 7.01 + 8.01)$$

$$= 8.00$$

Note that the exact solution is $y(2) = 8$.

## Program MILSIM

An algorithm for evaluating the equation $y' = f(x, y)$ using Milne-Simpson method is illustrated in Fig. 13.4. Program MILSIM shows the implementation of the algorithm in details. The program does the following:

1. Computes the starting points using fourth-order RK method
2. Predicts the function value by Milne's formula
3. Corrects the value obtained using Simpson's method
4. Writes the results

```
*-------------------------------------------------------------
      PROGRAM MILSIM                                          *
*-------------------------------------------------------------*
* Main program                                                *
*   This program solves the first order differential          *
*   equation y' = f(x,y) using Milne-Simpson method           *
*-------------------------------------------------------------*
* Functions invoked                                           *
*      F,INT                                                  *
*-------------------------------------------------------------*
* Subroutines used                                            *
*      NIL                                                    *
*-------------------------------------------------------------*
* Variables used                                              *
*      X(1) - Initial value of independent variable           *
*      Y(1)   Initial value of dependent variable             *
*        XP - Point of solution                               *
*         N - Number of steps                                 *
*         H - Step-size                                       *
*         X - Array of independent variable                   *
*         Y - Array of dependent variable                     *
*-------------------------------------------------------------*
* Constants used                                              *
*      NIL                                                    *
*-------------------------------------------------------------
```

```
      REAL X,Y,H,XP,M1,M2,M3,M4,SUM1,SUM2,F
      INTEGER N,INT
      INTRINSIC INT
      EXTERNAL F
      DIMENSION X(10), Y(10)
* Read values
      WRITE(*,*) 'Input initial values of x and y'
      READ(*,*) X(1),Y(1)
      WRITE(*,*) 'Input x at which y is required'
      READ(*,*) XP
      WRITE(*,*) 'Input step-size h'
      READ(*,*) H
* Compute number of computations involved
      N = INT((XP-X(1))/H + 0.5)
* We need four starting points for Milne-Simpson method.
* Initial values form the first point.  Remaining three
* points are obtained using 4th order RK method
      WRITE(*,*)
      WRITE(*,*) 'INITIAL VALUES', X(1), Y(1)
      WRITE(*,*)
* Computing three points by RK method
      WRITE(*,*) 'THREE VALUES BY RK METHOD'
      DO 10 I = 1,3
         M1 = F(X(I),Y(I))
         M2 = F(X(I)+0.5*H, Y(I)+0.5*M1*H)
         M3 = F(X(I)+0.5*H, Y(I)+0.5*M2*H)
         M4 = F(X(I)+H, Y(I)+M3*H)
         X(I+1) = X(I)+H
         Y(I+1) = Y(I)+(M1+2.0*M2+2.0*M3+M4)*H/6.0
         WRITE(*,*) I, X(I+1), Y(I+1)
10    CONTINUE
      WRITE(*,*)
      WRITE(*,*) 'VALUES OBTAINED BY MILNE-SIMPSON METHOD'
      DO 20 I = 4, N
         F2 = F(X(I-2), Y(I-2))
         F3 = F(X(I-1), Y(I-1))
         F4 = F(X(I), Y(I))
* Predicted value of y (by Milne's formula)
         Y(I+1) = Y(I-3)+4.0*H/3.0*(2.0*F2-F3+2.0*F4)
```

```
          X(I+1) = X(I) + H
          F5 = F(X(I+1), Y(I+1))
*  Corrected value of Y (by Simpson's formula)
          Y(I+1) = Y(I-1) + H/3.0 *(F3+4.0*F4+F5)
          WRITE(*,*) I, X(I+1), Y(I+1)
20     CONTINUE
       WRITE(*,*)
       WRITE(*,*)'Value of Y at X  =', X(N+1),'  is',
     +        Y(N+1)
       WRITE(*,*)
       STOP
       END
* --------------- End of main MILSIM ------------------ *
* ----------------------------------------------------- *
* Function subprogram                                   *
* ----------------------------------------------------- *
       REAL FUNCTION F(X,Y)
       REAL X,Y
       F = 2.0 * Y/X
       RETURN
       END
* -------------- End of function F(X,Y) -------------- *
```

***Test Run Results***

```
     Input initial values of x and y
     1.0 2.0
     Input x at which y is required
     2.0
     Input step-size h
     0.125

     INITIAL VALUES   1.0000000    2.0000000
     THREE VALUES BY RK METHOD
                 1     1.1250000    2.5312380
                 2     1.2500000    3.1249770
                 3     1.3750000    3.7812150
     VALUES OBTAINED BY MILNE-SIMPSON METHOD
                 4     1.5000000    4.4999660
                 5     1.6250000    5.2812040
                 6     1.7500000    6.1249520
```

```
                7          1.8750000       7.0311890
                8          2.0000000       7.9999360
    Value of Y at X = 2.0000000 is 7.9999360
    Stop - Program terminated.
```

## Adams-Bashforth-Moulton Method

Another popular fourth-order predictor-corrector method is the Adams-Bashforth-Moulton multistep method. The predictor formula is known as *Adams-Bashforth predictor* and is given by

$$\boxed{y_{i+1} = y_i + \frac{h}{24}(55f_i - 59f_{i-1} + 37f_{i-2} - 9f_{i-3})} \qquad (13.45)$$

The corrector formula is known as *Adams-Moulton corrector* and is given by

$$\boxed{y_{i+1} = y_i + \frac{h}{24}(f_{i-2} - 5f_{i-1} + 19f_i + 9f_{i+1})} \qquad (13.46)$$

This pair of equations can be implemented using the procedure described for Milne-Simpson method.

### Example 13.11

Repeat Example 13.10 using Adams-Bashforth-Moulton method.

By Adams predictor formula

$$y_4^p = y_3 + \frac{h}{24}(55f_3 - 59f_2 + 37f_1 - 9f_0)$$

$$= 6.13 + \frac{0.25}{24}(55 \times 7.01 - 59 \times 6.00 + 37 \times 5.01 - 9 \times 4)$$

$$= 8.0146$$

$$f_4^p = \frac{2 \times 8.0146}{2} \; 8.0146$$

By Adams corrector formula

$$y_4^c = y_3 + \frac{h}{24}(f_1 - 5f_2 + 19f_3 + 9f_4^p)$$

$$= 6.13 + \frac{0.25}{24}(5.01 - 5 \times 6.00 + 19 \times 7.01 + 9 \times 8.0146)$$

$$= 8.0086$$

$y_4^c$(refined) = 8.0079

## 13.9 ACCURACY OF MULTISTEP METHODS

We know that for each differential equation, there is an optimum step size $h$. If $h$ is too large, accuracy diminishes and if it is too small, round-off errors would dominate and reduce the accuracy.

By computing the predicted and corrected values of $y_{i+1}$, we can estimate the size and sign of the error. Let us denote the predicted value by $y^{p}_{i+1}$. Similarly, denote the truncation error in predicted value by $E_{tp}$ and corrected value by $E_{tc}$. Then, we have,

$$E_{tp} = y - y^{p}_{i+1}$$

$$E_{tc} = y - y^{c}_{i+1}$$

where $y$ denotes the exact value of $y(x_{i+1})$. The difference between the error is

$$E_{tp} - E_{tc} = y^{c}_{i+1} - y^{p}_{i+1} \qquad (13.47)$$

A large difference indicates that the step size is too large. In such cases, we must reduce the size of $h$.

### Milne-Simpson Method

Both the Milne and Simpson formulae are of order $h^4$ and their error terms are of order $h^5$.

The truncation error in Milne's formula is

$$E_{tp} = \frac{28}{90}\, y^{(5)}\,(\theta_1)\, h^5$$

The truncation error in Simpson's formula is

$$E_{tc} = -\frac{1}{90}\, y^{(5)}\,(\theta_2) h^5$$

If we assume that $y^{(5)}(\theta_1) = y^{(5)}(\theta_2)$ then

$$\frac{E_{tp}}{E_{tc}} = -28$$

or,

$$E_{tp} = -28E_{tc}$$

Substituting this in Eq. (13.47) we obtain,

$$E_{tc} = -\frac{y^{c}_{i+1} - y^{p}_{i+1}}{29}$$

If the answer is required to a precision of $d$ decimal digits then

$$|E_{tc}| = \left|\frac{y_{i+1}^{c} - y_{i+1}^{p}}{29}\right| < 0.5 \times 10^{-d}$$

or

$$\boxed{\left|y_{i+1}^{c} - y_{i+1}^{p}\right| < 29/2 \times 10^{-d} = 15 \times 10^{-d}} \tag{13.48}$$

## Adams Method

The truncation error in Adams-Bashforth predictor is

$$E_{tp} = \frac{251}{720}\, y^{(5)}\,(\theta_1)\, h^5$$

and the truncation error in Adams-Moulton corrector is

$$E_{tc} = -\frac{19}{720}\, y^{(5)}\,(\theta_2) h^5$$

Then, assuming $y^{(5)}(\theta_1) = y^{(5)}(\theta_2)$, we get

$$E_{tp} = -\frac{251}{19}\, E_{tc}$$

Substituting in Eq. (13.47) results in

$$E_{tc} = -\frac{19}{270}\left(y_{i+1}^{c} - y_{i+1}^{p}\right)$$

For achieving an accuracy of $d$ decimal digits,

$$\boxed{\left|y_{i+1}^{c} - y_{i+1}^{p}\right| < 270/19 \times 0.5 \times 10^{-d} \approx 7 \times 10^{-d}} \tag{13.49}$$

According to Eqs (13.48) and (13.49), $h$ should be reduced until the difference between the corrected and predicted values is within the specified limit. It should be noted, however, that if the step length is changed during the calculation, it will be necessary to recalculate the starting points at the new step value.

## Modifiers

Using the error estimates, we can modify the estimates of $y_{i+1}^{c}$ before proceeding to the next stage. That is

$$y_{i+1} = y_{i+1}^{c} + E_{tc}$$

For Milne's method

$$y_{i+1} = y_{i+1}^{c} - \frac{1}{29}\left(y_{i+1}^{c} - y_{i+1}^{p}\right)$$

For Adams method

$$y_{i+1} = y_{i+1}^{c} - \frac{19}{270}\left(y_{i+1}^{c} - y_{i+1}^{p}\right)$$

### [illegible] 12

Solve the differential equation

$$y'(x) = -y^2$$

for $y(2.0)$ using the Milne-Simpson method with the application of modifier to the corrector. The first four points are given under

| $i$ | $x_i$ | $y_i = y(x_i)$ |
|---|---|---|
| 0 | 1.0 | 1.0000000 |
| 1 | 1.2 | 0.8333333 |
| 2 | 1.4 | 0.7142857 |
| 3 | 1.6 | 0.6250000 |

To estimate $y(2.0)$ with $h = 0.2$, we need two iterations.

$$\begin{aligned} f_i &= -y_i^2 \\ f_1 &= -0.6944444 \\ f_2 &= -0.5102040 \\ f_3 &= -0.3906250 \end{aligned}$$

*Iteration 1*

$$y(1.8) = y_4^p = y_0 + \frac{4h}{3}(2f_1 - f_2 + 2f_3)$$

$$= 1.0 + \frac{4 \times 0.2}{3}(-1.13888889 + 0.5102040 - 0.7812500)$$

$$= 0.5573506$$

$$f_4^p = -\left(y_4^p\right)^2 = -0.3106396$$

$$y_4^c = y_2 + \frac{h}{3}\left(f_2 + 4f_3 + f_4^p\right)$$

$$= 0.7142857 + \frac{0.2}{3}(-0.5102040 - 1.56250 - 0.3106396)$$

$$= 0.5553961$$

$$\text{Modifier } E_{tc} = -\frac{y_4^c - y_4^p}{29} = 0.0000674$$

$$\text{Modified } y_4^c = y_4^c + E_{tc} = 0.5554635$$

*Iteration 2*

$$f_4 = -\left(y_4^c\right)^2 = -0.3085397$$

$$y(2.0) = y_5^p = y_1 + \frac{4h}{3}\,(2f_2 - f_3 + 2f_4)$$

$$= 0.5008366$$

$$f_5^p = -0.2508373$$

$$y_5^c = y_3 + \frac{h}{3}\left(f_3 + 4f_4 + f_5^p\right)$$

$$= 0.4999585$$

$$\text{Modifier } E_{tc} = 0.0000303$$

$$\text{Modified } y_5^c = 0.4999888$$

$$\text{Exact answer} = 0.5$$

$$\text{Error} = 0.0000112$$

## 13.10 SYSTEMS OF DIFFERENTIAL EQUATIONS

Mathematical models of many applications involve a system of several first-order differential equations. They may be represented as follows:

$$\frac{dy_1}{dx} = f_1(x, y_1, y_2, \ldots, y_m), \qquad y_1(x_0) = y_{10}$$

$$\frac{dy_2}{dx} = f_2(x, y_1, y_2, \ldots, y_m), \qquad y_2(x_0) = y_{20}$$

$$\vdots \tag{13.50}$$

$$\frac{dy_m}{dx} = f_m(x, y_1, y_2, \ldots, y_m), \qquad y_m(x_0) = y_{m0}$$

These equations should be solved for $y_1(x)$, $y_2(x)$, ..., $y_m(x)$ over interval $(a, b)$.

These equations can be solved by any of the methods discussed in this chapter. At each stage, all the equations are solved before proceeding to the next stage. For example, if $h = 0.5$ and $a = x_0 = 0$, then we must evaluate $y_1(0.5)$, $y_2(0.5)$,..., $y_m(0.5)$ before preceding to the stage $h = 1.0$. Let us consider a system of two equations for the purpose of illustration.

$$y_1'(x) = f_1(x, y_1, y_2), \qquad y_1(x_0) = y_{10}$$

$$y_2'(x) = f_2(x, y_1, y_2), \qquad y_2(x_0) = y_{20}$$

Assume that we want to use the Heun's method. The first stage would involve the following calculations.

$$m_1(1) = f_1(x_0, y_{10}, y_{20})$$

$$m_1(2) = f_2(x_0, y_{10}, y_{20})$$

$$m_2(1) = f_1(x_0 + h, y_{10} + hm_1(1), \qquad y_{20} + hm_1(2))$$

$$m_2(2) = f_2(x_0 + h, y_{10} + hm_1(1), \qquad y_{20} + hm_1(2))$$

$$m(1) = \frac{m_1(1) + m_2(1)}{2}$$

$$m(2) = \frac{m_1(2) + m_2(2)}{2}$$

$$y_1(x_1) = y_1(1) = y_1(x_0) + m(1)h = y_{10} + m(1)h$$

$$y_2(x_1) = y_2(1) = y_2(x_0) + m(2)h = y_{20} + m(2)h$$

The next stage uses $y_1(1)$ and $y_2(1)$ as initial values and, by following similar procedure, $y_1(2)$ and $y_2(2)$ are obtained.

### Example 13.13

Given the equations

$$\frac{dy_1}{dx} = x + y_1 + y_2, \qquad y_1(0) = 1$$

$$\frac{dy_2}{dx} = 1 + y_1 + y_2, \qquad y_2(0) = -1$$

estimate the values of $y_1(0.1)$ and $y_2(0.1)$ using Heun's method.

Given $x_0 = 0, \quad y_{10} = 1, \quad y_{20} = -1$

$$m_1(1) = f_1(x_0, y_{10}, y_{20}) = 0 + 1 - 1 = 0$$

$$m_1(2) = f_2(x_0, y_{10}, y_{20}) = 1 + 1 - 1 = 1$$

$$\begin{aligned} m_2(1) &= f_1(x_0 + h, y_{10} + hm_1(1), y_{20} + hm_1(2)) \\ &= f_1(0.1, 1 + 0.1 \times 0, -1 + 0.1 \times 1) \\ &= f_1(0.1, 1, -0.9) \\ &= 0.1 + 1 - 0.9 = 0.2 \end{aligned}$$

$$\begin{aligned} m_2(2) &= f_2(x_0 + h, y_{10} + hm_1(1), y_{20} + hm_1(2)) \\ &= f_2(0.1, 1, -0.9) \\ &= 1 + 1 - 0.9 = 1.1 \end{aligned}$$

$$m(1) = \frac{m_1(1) + m_2(1)}{2} = 0.1$$

$$m(2) = \frac{m_1(2) + m_2(2)}{2} = 1.05$$

$$y_1(0.1) = y_1(0) + hm(1) = 1 + (0.1)(0.1) = 1.01$$

$$y_2(0.1) = y_2(0) + h\,m(2) = -1 + (0.1)(1.05) = -0.895$$

## 13.11 HIGHER-ORDER EQUATIONS

We have seen in the introductory section of this chapter that many problems involve the solution of higher-order differential equations. A higher-order differential equation is in the form

$$\frac{d^m y}{d^m x} = f\left(x, y, \frac{dy}{dx}, \frac{d^2y}{dx^2}, \ldots, \frac{d^{m-1}y}{dx^{m-1}}\right) \tag{13.51}$$

with $m$ initial conditions given as

$$y(x_0) = a_1, \quad y'(x_0) = a_2, \ldots, y^{m-1}(x_0) = a_m$$

We can replace Eq. (13.51) by a system of first-order equations as follows:

Let us denote

$$y = y_1, \quad \frac{dy}{dx} = y_2, \quad \frac{d^2y}{dx^2} = y_3, \ldots, \quad \frac{d^{m-1}y}{dx^{m-1}} = y_m$$

Then,

$$\boxed{\begin{array}{ll} \dfrac{dy_1}{dx} = y_2 & y_1(x_0) = y_{10} = a_1 \\ \dfrac{dy_2}{dx} = y_3 & y_2(x_0) = y_{20} = a_2 \\ \cdot & \\ \cdot & \\ \cdot & \\ \dfrac{dy_{m-1}}{dx} = y_m & y_{m-1}(x_0) = y_{m-1,0} = a_{m-1} \\ \dfrac{dy_m}{dx} = f(x, y_1, y_2, \ldots, y_m) & y_m(x_0) = y_{m0} = a_m \end{array}} \tag{13.52}$$

This system is similar to the system of first-order Eq. (13.50) with the conditions

$$f_i = y_{i+1}, \qquad i = 1, 2, \ldots, m-1$$

$$f_m = f(x, y_1, y_2, \ldots, y_m)$$

and, therefore, can be solved using the procedure discussed in the previous section.

### Example 13.14

Solve the following equation for $y(0.2)$

$$\frac{d^2y}{dx^2}+2\frac{dy}{dx}-3y=6x$$

given $y(0) = 0$, $y'(0) = 1$. Usc Heun's method

$$\frac{d^2y}{dx^2}=6x+3y-2\frac{dy}{dx}$$

Let $$y=y_1,\ \frac{dy}{dx}=y_2$$

Then,

$$\frac{dy_1}{dx}=y_2, \qquad y_{10}=0$$

$$\frac{dy_2}{dx}=6x+3y_1-2y_2, \qquad y_{20}=1$$

Let $h = 0.2$

$$m_1(1)=y_{20}=1$$
$$m_1(2)=6x_0+3y_{10}-2y_{20}=-2$$
$$m_2(1)=y_{20}+hm_1(2)=1+(0.2)(-2)=0.6$$
$$m_2(2)=6(0.2)+3(0+0.2(1))-2(0.6)=1.2+0.6-1.2=0.6$$
$$m(1)=\frac{1+0.6}{2}=0.8$$
$$m(2)=\frac{-2+0.6}{2}=-0.7$$
$$y_1(0.2)=y_1(0)+0.2m(1)=0+0.2\times0.8=0.16$$
$$y_2(0.2)=y_2(0)+0.2m(2)=1-(0.2)(0.7)=0.86$$
$$y(x) \text{ at } x=0.2=0.16$$
$$y'(x) \text{ at } x=0.2=0.86$$

## 13.12 SUMMARY

We encounter differential equations in many different forms while attempting to solve real life problems in science and engineering. The most common form of differential equations is known as ordinary differential equation. In this chapter, we considered various methods of numerical solution of ordinary differential equations. They include

- Taylor series method
- Euler's method
- Heun's method
- Polygon method
- Runge-Kutta method
- Milne-Simpson method
- Adams-Bashforth-Moulton method

We also discussed the accuracy (and techniques of improving the accuracy) of these methods. Finally, we discussed the solution of a system of differential equations as well as higher-order equations.

We presented FORTRAN programs and their test results for the following methods:

- Euler's method
- Heun's method
- Polygon method
- Runge-Kutta method
- Milne-Simpson method

## Key Terms

*Adams-Bashforth predictor*
*Adams-Moulton corrector*
*Boundary-value problem*
*Corrector*
*Differential equations*
*Euler's method*
*Global truncation error*
*Heun's method*
*Initial-value problem*
*Kirchhoff's law*
*Law of cooling*
*Law of motion*
*Local truncation error*
*Midpoint method*
*Milne's formula*
*Milne-Simpson method*
*Multistep methods*
*One-step methods*
*Ordinary differential equations*
*Partial differential equation*
*Picard's method*
*Polygon method*
*Predictor*
*Predictor-corrector method*
*Radioactive decay*
*Runge-Kutta method*
*Semi-numeric method*
*Simple harmonic motion*
*Simpson's formula*
*Starting values*
*Taylor series method*

## REVIEW QUESTIONS

1. What is a differential equation? Give two real-life examples of application of differential equations.
2. Distinguish between ordinary and partial differential equations.

3. State the degree and order of the following differential equations:
   (a) $(y'')^2 + 7y' = 0$
   (b) $y''' + 5(y')^2 = 1$
   (c) $y'' - y = 0$
   (d) $y' + ay^2 = 0$
   (e) $y'' + 3y' - 2y = x^2$
   (f) $y(y')^2 - 2xy' + y = 0$
   (g) $xy' - (x + 1)y = 0$
   (h) $y^2(y')^2 + xy\, y' - 2x^2 = 0$
   (i) $(y'')^2 + y^2 = 0$
4. What is a nonlinear differential equation? State which of the equations given in Question 3 are nonlinear.
5. What is an initial-value problem? How is it different from a boundary-value problem?
6. Why do we need to use numerical computing techniques to solve differential equations?
7. State the basic two approaches used in estimating the solution of differential equations. How are they different?
8. Describe how Taylor's theorem of expansion can be used to solve a differential equation.
9. What are the limitations of Taylor's series method?
10. State the formula of Picard's method to solve the differential equation of type

$$\frac{dy}{dx} = f(x, y)$$

    What are its limitations?
11. State the formula of Euler's method. Illustrate its concept graphically.
12. Comment on the accuracy of Euler's method.
13. Illustrate Heun's method of solution graphically.
14. How does the accuracy of Heun's method compare with that of Euler's method?
15. Heun's method is an improved version of Euler's method. Comment.
16. Why is Heun's method classified as one-step predictor-corrector method?
17. Why is the polygon method called the midpoint method? Illustrate graphically.
18. Describe the basic concept employed in Runge-Kutta methods.
19. What is meant by an $r$-order Runge-Kutta method? What is the order of the following methods?
    (a) Euler's method
    (b) Heun's method
    (c) Polygon method

20. What are multistep methods?
21. What are the merits and demerits of multistep methods?
22. State the formulae used in Milne-Simpson method. Describe the implementation scheme of these formulae.
23. [illegible] predictor and corrector formulae used in Adams [illegible]
24. [illegible]? How is it used to improve the accuracy of multistep methods?
25. A high-order differential equation can be solved by replacing it by a system of first-order equations. Discuss.

## REVIEW EXERCISES

1. Use Taylor's expansion (with terms up to $x^3$) to solve the following differential equations:

(a) $\dfrac{dy}{dx} = x + y + xy, \qquad y(0) = 1$

for $x = 0.25$ and $0.5$.

(b) $\dfrac{dy}{dx} = y(x^2 - 1), \qquad y(0) = 1$

for $x = 1.0$, $1.5$ and $2.0$

(c) $\dfrac{dy}{dx} = x + y, \qquad y(0) = 1$

for $x = 0.1$ and $0.5$

(d) $\dfrac{dy}{dx} = \dfrac{2x}{y} - xy, \qquad y(0) = 1$

for $x = 0.25$ and $0.5$

(e) $\dfrac{dy}{dx} = x^2y^2, \qquad y(1) = 0$

for $x = 2.0$ and $3.0$

(f) $\left(\dfrac{dy}{dx}\right)^2 = xy, \qquad y(0) = 1$ and $y'(0) = 1$

for $x = 0.2$ and $0.4$

(g) $10\dfrac{d^2y}{dt^2} + \left(\dfrac{dx}{dt}\right)^2 + 6x = 0, \qquad x(0) = 1$ and $x'(0) = 0$

for $t = 2$ and $5$.

2. Solve the following equations by Picards method and estimate $y$ at $x = 0.25$ and $0.5$:

   (a) $\dfrac{dy}{dx} = x + (x + 1)y, \quad y(0) = 1$

   (b) $\dfrac{dy}{dx} = x^2y - y, \quad y(0) = 1$

   (c) $\dfrac{dy}{dx} = x + y, \quad y(0) = 1$

   (d) $\dfrac{dy}{dx} = x^2y^2, \quad y(1) = 0$

   (e) $\dfrac{dy}{dx} = \dfrac{x}{y}, \quad y(0) = 1$

3. Use the simple Euler's method to solve the following equations for $y(1)$ using $h = 0.5$, $0.25$ and $0.1$.

   (a) $y' = 2xy, \quad y(0) = 1$

   (b) $y' = x^2 + y^2, \quad y(0) = 2$

   (c) $y' = \dfrac{-y}{2y+1}, \quad y(0) = 1$

   (d) $y' = \dfrac{x}{y}, \quad y(0) = 1$

   (e) $y' = x + y + xy, \quad y(0) = 1$

4. Solve the differential equation

$$\frac{dy}{dx} = x + y, \quad y(0) = 1$$

by the simple Euler's method to estimate $y(1)$ using $h = 0.5$ and $h = 0.25$. Compute errors in both the cases. How do they compare? Also, compare your results with the exact answer given the analytical solution as

$$y(x) = 2e^x - x - 1$$

5. Use Heun's method with $h = 0.5$ and $h = 0.25$ to solve the equations in Exercise 3 for $y(1)$. How do the results compare with these obtained using simple Euler's method.
6. Repeat Exercise 4 using the Polygon method instead of simple Euler's method. Analyse the results critically.
7. Use the polygon method to solve some of the equation in Exercise 3.
8. Use the classical RK method to estimate $y(0.5)$ of the following equations with $h = 0.25$.

   (a) $\dfrac{dy}{dx} = x + y, \quad y(0) = 1$

(b) $\dfrac{dy}{dx} = \dfrac{x}{y}$, $\quad y(0) = 1$

(c) $\dfrac{dy}{dx} = y \cos x$, $\quad y(0) = 1$

(d) $\dfrac{dy}{dx} = y + \sin x$, $\quad y(0) = 2$

(e) $\dfrac{dy}{dx} = y + \sqrt{y}$, $\quad y(0) = 1$

9. Solve the following initial value problems for $x = 1$ using the fourth-order Milne's method.

   (a) $\dfrac{dy}{dx} = y - x^2$, $\quad y(0) = 1$

   Use a step size of 0.25 and fourth-order Runge-Kutta method to predict the starting values.

10. Repeat Exercise 9 using Adam's method instead of Milne's method. Compare the results.

11. Repeat Exercise 10 and 11 with the application of modifier to the corrector. Compare the results.

12. Solve the pair of simultaneous equations

$$\frac{dy_1}{dx} = y_2 \qquad y_1(0) = 0$$

$$\frac{dy_2}{dx} = y_1 y_2 + x^2 + 1 \qquad y_2(0) = 0$$

   to estimate $y_1(0.2)$ and $y_2(0.2)$ using any method of your choice.

13. Solve the following equation for $y(0.2)$:

$$10\frac{d^2y}{dx^2} + \left(\frac{dy}{dx}\right)^2 + 6x = 0, \qquad y(0) = 1, \qquad y'(0) = 0$$

   Use Heun's method.

14. The general equation relating to current $i$, voltage $V$, resistance $R$, and inductance $L$ of a serial electric circuit is given by

$$L\frac{di}{dt} + iR = V$$

   Find the value of current after 2 seconds, if resistance $R = 20$ ohms, inductance $L = 50$ H and voltage $V = 240$ volts. Current $I = 0$ when $t = 0$.

15. A tank contains a solution which is made dissolving 50 kg of salt in 100 gallons of water. A more concentrated solution of 3 kg of salt per gallon of water is pumped into the tank at a rate of 4 gallons per minute. The solution (which is stirred continuously to keep it uniform) is pumped out at a rate of 3 gallons per minute.
    Find the amount of salt in the tank at $t = 15$ minutes. Note that initially at $t = 0$, the amount of salt $x = 50$ kg.

    **Hint:** If $x$ represents the amount of salt in the tank at time $t$, $\frac{dx}{dt}$ will represent the change in the amount of salt.

16. An object with a mass of 10 kg is falling under the influence of earth gravity. Find its velocity after 5 seconds if it starts from the rest. The object experiences a retarding force equal to 0.25 of its velocity.
    **Hint:** The relationship between the various forces is given by

    $$\text{mass} \times \frac{dv}{dt} = \text{mass} \times \text{gravity} - \text{retarding force}$$

    where $v$ is velocity at time $t$.

17. A body of mass 2 kg is attached to a spring with a spring constant of 10. The differential equation governing the displacement of the body $y$ and time $t$ is given by

    $$\frac{d^2 y}{dt^2} + 2\frac{dy}{dt} + 5y = 0$$

    Find the displacement $y$ at time $t = 1.5$ given that $y(0) = 2$ and $y'(0) = -4$.

## PROGRAMMING PROJECTS

1. Rewrite the program RUNGE4 such that a subprogram implements the fourth order Runge-Kutta method and a main program receives input information, drives the subprogram to compute the solution, and prints the required output information.
2. Modify the program MILSIM to incorporate the following changes:
   (a) Computing the starting points by Runge-Kutta method using a subprogram
   (b) Implementing Milne-Simpson algorithm by a subprogram
   (c) Applying the modifier to the corrector with the help of a subprogram.
3. Develop a user-friendly modular program as suggested in Project 2 for the fourth-order Adams method with modifiers.
4. Develop a user-friendly program for solving systems of differential equations using Euler's method or Heun's method.
5. Repeat Project 4, but use the fourth-order Runge-Kutta method.

CHAPTER 14

# Boundary Value and Eigenvalue Problems

## 14.1 NEED AND SCOPE

We have seen that we require $m$ conditions to be specified in order to solve an $m$-order differential equation. In the previous chapter, all the $m$ conditions were specified at one point, $x = x_0$, and, therefore, we call this problem as an *initial-value problem*. It is not always necessary to specify the conditions at one point of the independent variable. They can be specified at different points in the interval $(a, b)$ and, therefore, such problems are called the *boundary value problems*. A large number of problems fall into this category.

In solving initial value problems, we move in steps from the given initial value of $x$ to the point where the solution is required. In case of boundary value problems, we seek solutions at specified points within the domain of given boundaries, for instance, given

$$\frac{d^2 y}{dx^2} = f(x, y, y') \qquad y(a) = y_a, \qquad y(b) = y_b \qquad (14.1)$$

we are interested in finding the values of $y$ in the range $a \leq x \leq b$.

There are two popular methods available for solving the boundary value problems. The first one is known as the *shooting method*. This method makes use of the techniques of solving initial value problems. The second one is called the *finite difference method* which makes use of the finite difference equivalents of derivatives.

Some boundary value problems, such as study of vibrating systems, structure analysis, and electric circuit system analysis, reduce to a system of equations of the form

$$\mathbf{A}\mathbf{x} = \lambda \mathbf{x} \qquad (14.2)$$

Such problems are called the *eigenvalue problems*. We need to determine the values of $\lambda$ and vector $x$ which satisfy the Eq. (14.2). We have two simple methods available to solve this type of problems.

1. Polynomial method
2. Power method

In this chapter, we consider these two special categories of problems and discuss the following methods to solve them:

1. Shooting method
2. Finite difference method
3. Polynomial method
4. Power method

## SHOOTING METHOD

This method is called the *shooting method* because it resembles an artillery problem. In this method, the given boundary value problem is first converted into an equivalent initial value problem and then solved using any of the methods discussed in the previous chapter. The approach is simple. Consider the equation

$$y'' = f(x, y, y') \qquad y(a) = A, \qquad y(b) = B$$

By letting $y' = z$, we obtain the following set of two equations:

$$y' = z$$
$$z' = f(x, y, z)$$

In order to solve this set as an initial value problem, we need two conditions at $x = a$. We have one condition $y(a) = A$ and, therefore, require another condition for $z$ at $x = a$. Let us assume that $z(a) = M_1$, where $M_1$ is a "guess". Note that $M_1$ represents the slope $y'(x)$ at $x = a$. Thus, the problem is reduced to a system two first-order equations with the initial conditions

$$\begin{aligned} y' &= z & y(a) &= A \\ z' &= f(x, y, z) & z(a) &= M_1\ (= y'\ (\mathrm{a})) \end{aligned} \qquad (14.3)$$

Equation (14.3) can be solved for $y$ and $z$ using any one-step method using steps of $h$, until the solution at $x = b$ is reached. Let the estimated value of $y(x)$ at $x = b$ be $B_1$. If $B_1 = B$, then we have obtained the required solution. In practice, it is very unlikely that our initial guess $z(a) = M_1$ is correct.

If $B_1 \neq B$, then we obtain the solution with another guess, say $z(a) = M_2$. Let the new estimate of $y(x)$ at $x = b$ be $B_2$ (see Fig. 14.1). If $B_2$ is not equal to $B$, then the process may be continued until we obtain the correct estimate of $y(b)$. However, the procedure can be accelerated by using an improved guess for $z(a)$ after the estimates of $B_1$ and $B_2$ are obtained.

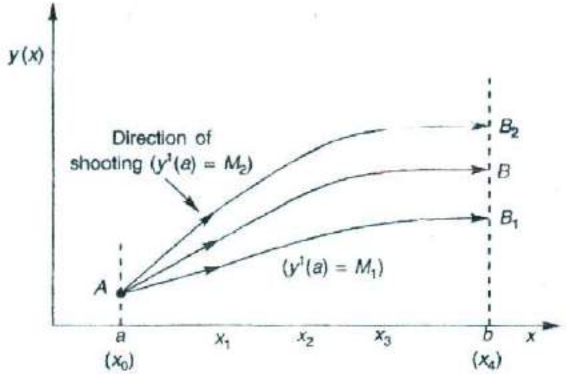


Fig. 14.1 Illustration of shooting method

Let us assume that $z(a) = M_3$ leads to the value $y(b) = B$. If we assume that the values of $M$ and $B$ are linearly related, then

$$\frac{M_3 - M_2}{B - B_2} = \frac{M_2 - M_1}{B_2 - B_1}$$

Then

$$M_3 = M_2 + \frac{B - B_2}{B_2 - B_1} \times (M_2 - M_1)$$

$$= M_2 - \frac{B_2 - B}{B_2 - B_1} \times (M_2 - M_1) \tag{14.4}$$

Now with $z(a) = M_3$, we can again obtain the solution of $y(x)$.

### Example 14.1

Using shooting method, solve the equation

$$\frac{\mathrm{d}^2 y}{\mathrm{d}x^2} = 6x, \qquad y(1) = 2, \qquad y(2) = 9$$

in the interval (1,2)

By transformation we obtain the following:

$$\frac{\mathrm{d}y}{\mathrm{d}x} = z \qquad y(1) = 2$$

$$\frac{\mathrm{d}z}{\mathrm{d}x} = 6x$$

Let us assume that $z(1) = y'(1) = 2(M_1)$. Applying Hei n's method, we obtain the solution as follows:

*Iteration 1*

$$h = 0.5$$

$$x_0 = 1, \quad y(1) = y_0 = 2, \quad z(1) = z_0 = 2$$

$$m_1(1) = z_0 = 2$$

$$m_1(2) = 6x_0 = 6$$

$$m_2(1) = z_0 + hm_1(2) = 2 + 0.5(6) = 5$$

$$m_2(2) = 6(x_0 + h) = 6(1.5) = 9$$

$$m(1) = \frac{m_1(1) + m_2(1)}{2} = \frac{2+5}{2} = 3.5$$

$$m(2) = \frac{m_1(2) + m_2(2)}{2} = \frac{6+9}{2} = 7.5$$

$$y(x_1) = y(1.5) = (1) + m(1)h = 2 + 3.5 \times 0.5 = 3.75$$

$$z(x_1) = z(1.5) = z(1) + m(2)h = 2 + 7.5 \times 0.5 = 5.75$$

*Iteration 2*

$$h = 0.5$$

$$x_1 = 1.5, \quad y_1 = 3.75, \quad z_1 = 5.75$$

$$m_1(1) = z_1 = 5.75$$

$$m_1(2) = 6x_1 = 9$$

$$m_2(1) = z_1 + hm_1(2) = 5.75 + 0.5 \times 9 = 10.25$$

$$m_2(2) = 6(x_1 + h) = 12$$

$$m(1) = \frac{5.75 + 10.25}{2} = 8$$

$$m(2) = \frac{9 + 12}{2} = 10.5$$

$$y(x_2) = (2) = y(1) + m(1)h = 3.75 + 8 \times 0.5 = 7.75$$

This gives $B_1 = 7.75$ which is less than $B = 9$

Now, let us assume $z(1) = y'(1) = 4(M_2)$ and again estimate $y(2)$.

*Iteration 1*

$$h = 0.5$$

$$x_0 = 1, \quad y_0 = 2, \quad z_0 = 4$$

$$m_1(1) = z_0 = 4$$

$$m_1(2) = 6x_0 = 6$$

$$m_2(1) = z_0 + hm_1(2) = 4 + 0.5 \times 6 = 7$$

$$m_2(2) = 6(x_0 + h) = 6(1.5) = 9$$

$$m(1) = \frac{4 + 7}{2} = 5.5$$

$$m(2) = \frac{6+9}{2} = 7.5$$

$$y(x_1) = y(1.5) = 2 + 5.5 \times 0.5 = 4.75$$

$$z(x_1) = z(1.5) = 4 + 7.5 \times 0.5 = 7$$

*Iteration 2*

$$h = 0.5$$

$$x_1 = 1.5, \quad y_1 = 4.75, \quad z_1 = 7.75$$

$$m_1(1) = z_1 = 7.75$$

$$m_1(2) = 6x_1 = 9$$

$$m_2(1) = z_1 + hm_1(2) = 7.75 + 0.5 \times 9 = 12.25$$

$$m_2(2) = 6(x_1 + h) = 12$$

$$m(1) = \frac{7.75 + 12.25}{2} = 10$$

$$m(2) = \frac{9+12}{2} = 10.5$$

This gives $B_2 = 9.75$ which is greater than $B = 9$.

Now, let us have the third estimate of $z(1) = M_3$ using the relationship (14.4)

$$M_3 = M_2 - \frac{B_2 - B}{B_2 - B_1} \times (M_2 - M_1)$$

$$= 4 - \frac{(9.75 - 9)}{(9.75 - 7.75)}(4 - 2)$$

$$= 4 - 0.75 = 3.25$$

The new estimate for $z(1) = y'(1) = 3.25$

*Iteration 1*

$$h = 0.5$$

$$x_0 = 1, \quad y_0 = 2, \quad z_0 = 3.25$$

$$m_1(1) = z_0 = 3.25$$

$$m_1(2) = 6x_0 = 6$$

$$m_2(1) = z_0 + hm_1(2) = 3.25 + 0.5 \times 6 = 6.25$$

$$m_2(2) = 6(x_0 + h) = 9$$

$$m(1) = \frac{3.25 + 6.25}{2} = 4.75$$

$$m(2) = \frac{6+9}{2} = 7.5$$

$$y(1.5) = 2 + 4.75 \times 0.5 = 4.375$$
$$z(1.5) = 3.25 + 7.5 \times 0.5 = 7$$

*Iteration 2*

$$h = 0.5$$
$$x_1 = 1.5, \qquad y_1 = 3.75, \qquad z_1 = 7$$
$$m_1(1) = z_1 = 7$$
$$m_1(2) = 6x_1 = 6 \times 1.5 = 9$$
$$m_2(1) = z_1 + hm_1(2) = 7 + 0.5 \times 9 = 11.5$$
$$m_2(2) = 6(z_1 + h) = 12$$
$$m(1) = \frac{7 + 11.25}{2} = 9.25$$
$$m(2) = \frac{9 + 12}{2} = 10.5$$
$$y(2) = 4.375 + 9.25 \times 0.5 = 9$$

The solution is $y(1) = 2$, $y(1.5) = 4.375$, $y(2) = 9$ The exact solution is $y(x) = x^3 + 1$ and therefore $y(1.5) = 4.375$.

The sequence of procedures for implementir the shooting method is given in Algorithm 14.1.

**Shooting Method**

1. Convert the problem into an initial-value problem.
2. Initialise the variables including two guesses at the initial slope.
3. Solve the equations with these guesses using either a one-step or a multistep method.
4. Interpolate from these results to find an improved value of the slopes obtained.
5. Repeat the process until a specified accuracy in the final function value is obtained (or until a limit to the number of iterations is reached).

**Algorithm 14.1**

## 14.3 FINITE DIFFERENCE METHOD

In this method, the derivatives are replaced by their finite difference equivalents, thus converting the differential equation into a system of algebraic equations. For example, we can use the following "central difference" approximations:

$$y_i' = \frac{y_{i+1} - y_{i-1}}{2h} \tag{14.5}$$

$$y_i'' = \frac{y_{i+1} - 2y_i - y_{i-1}}{h^2} \tag{14.6}$$

These are second-order equations and the accuracy of estimates can be improved by using higher-order equations.

The given interval $(a, b)$ is divided into $n$ subintervals, each of width $h$. Then

$$x_i = x_0 + ih = a + ih$$
$$y_i = y(x_i) = y(a + ih)$$
$$y_0 = y(a)$$
$$y_n = y(a + nh) = y(b)$$

This is illustrated in Fig. 14.2. The difference equation is written for each of the internal points $i = 1, 2, ..., n - 1$. If the *DE* is linear, this will result with $(n - 1)$ unknowns $y_1, y_2, ..., y_{n-1}$. We can solve for these unknowns using any of the elimination methods.

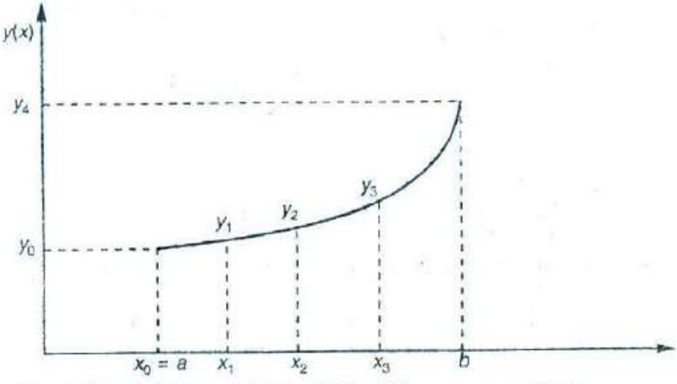


**Fig. 14.2** Solution of DE by finite difference method

Note that smaller the size of $h$, more the subintervals and, therefore, more are the equations to be solved. However, a smaller $h$ yields better estimates.

## Example 14.2

Given the equation

$$\frac{d^2y}{dx^2} = e^{x^2} \qquad \text{with} \qquad y(0) = 0, \qquad y(1) = 0$$

estimate the values of $y(x)$ at $x = 0.25$, 0.5 and 0.75.

We know that

$$\begin{aligned}
y_0 &= y(0) = 0\\
y_1 &= y(0.25)\\
y_2 &= y(0.5)\\
y_3 &= y(0.75)\\
y_4 &= y(1) = 0\\
h &= 0.25
\end{aligned}$$

$$y_i'' = \frac{y_{i+1} - 2y_i + y_{i-1}}{h^2} = e^{x^2}$$

*i = 1, x = 0.25*

$$y_1'' = \frac{y_2 - 2y_1 + y_0}{0.0625} = e^{(0.25)^2} = 1.0645$$

$$\boxed{y_2 - 2y_1 + y_0 = 0.0665} \qquad (1)$$

*i = 2, x = 0.50*

$$y_2'' = \frac{y_3 - 2y_2 + y_1}{0.0625} = e^{(0.5)^2} = 1.2840$$

$$\boxed{y_3 - 2y_2 + y_1 = 0.0803} \qquad (2)$$

*i = 3, x = 0.75*

$$y_3'' = \frac{y_4 - 2y_3 + y_2}{0.0625} = e^{(0.75)^2} = 1.7551$$

$$\boxed{y_4 - 2y_3 + y_2 = 0.1097} \qquad (3)$$

Letting $y_0 = 0$ and $y_4 = 0$, we have the following system of three equations.

$$\begin{aligned}
-2y_1 + y_2 &= 0.0665\\
y_1 - 2y_2 + y_3 &= 0.0803\\
y_2 - 2y_3 &= 0.1097
\end{aligned}$$

Solution of these equations results in

$$\begin{aligned}
y_1 &= y(0.25) = -0.1175\\
y_2 &= y(0.50) = -0.1684\\
y_3 &= y(0.75) = -0.1391
\end{aligned}$$

The major steps of finite difference method are given in Algorithm 14.2.

### Finite Difference Method

1. Divide the given interval into $n$ subinterval
2. At each point of $x$, obtain difference equation using a suitable difference formula. This will result in $(n - 1)$ equations with $(n - 1)$ unknowns, $y_1, y_2, ..., y_n - 1$.
3. Solve for $y_i$, $i = 1, 2, ...., n - 1$ using any of the standard elimination methods.

Algorithm 14.2

## 14.4 SOLVING EIGENVALUE PROBLEMS

As mentioned earlier, some boundary value problems, when simplified, may result in a set of homogeneous equation of the type

$$
\begin{aligned}
(a_{11} - \lambda)x_1 + a_{12}x_2 + ... + a_1nx_n &= 0 \\
a_{21}x_1 + (a_{22} - \lambda)x_2 + ... + a_{2n}x_n &= 0 \\
\cdots \\
\cdots \\
a_{n1}x_1 + a_{n2}x_2 + ... + (a_{nn} - \lambda)x_n &= 0
\end{aligned}
\tag{14.7}
$$

where $\lambda$ is a scalar constant. Equation (14.7) may be expressed as

$$[\mathbf{A} - \lambda \mathbf{I}]\,[\mathbf{X}] = 0 \tag{14.8}$$

where **I** is the *identity matrix* and $[\mathbf{A} - \lambda\mathbf{I}]$ is called the *characteristic matrix* of the coefficient matrix **A**.

The homogeneous Eq. (14.8) will have a non-trivial solution if, and only if, the characteristic matrix is singular. That is, the matrix $[\mathbf{A} - \lambda\mathbf{I}]$ is not invertible. Then, we have

$$|\mathbf{A} - \lambda\mathbf{I}| \tag{14.9}$$

Expansion of the determinant will result in a polynomial of degree $n$ in $\lambda$.

$$\lambda^n - p_1\lambda^{n-1} p_2\lambda^{n-2} - .... - p_{n-1}\lambda - p_n = 0 \tag{14.10}$$

Equation (14.10) will have $n$ roots $\lambda_1, \lambda_2, ..., \lambda_n$. The equation is known as the *characteristic polynomial* (or *characteristic equation*) and the roots are known as the *eigenvalues or characteristic values* of the matrix **A**. The solution vectors $X_1, X_2, ..., X_n$ corresponding to the eigenvalues $\lambda_1, \lambda_2, ..., \lambda_n$ are called the *eigenvectors*.

The roots representing the eigenvalues may be real distinct, real repeated, or complex, depending on the nature of the coefficient matrix **A**. The coefficients $p_i$ of the characteristic polynomial are functions of the matrix elements $a_{ij}$ and must be determined before the polynomial can be used.

Example 14.3 illustrates the procedure of evaluation of eigenvalues and eigenvectors of a simple system.

**Example 14.3**

Find the eigenvectors of the following system:

$$8x_1 - 4x_2 = \lambda x_1$$
$$2x_1 + 2x_2 = \lambda x_2$$

The characteristic equation of the given system is

$$\begin{vmatrix} 8-\lambda & -4 \\ 2 & 2-\lambda \end{vmatrix} = 0$$

That is

$$(8 - \lambda)(2 - \lambda) + 8 = 0$$

or

$$\lambda^2 - 10\lambda + 8 = 0$$

The roots are

$$\lambda_1 = 6$$
$$\lambda_2 = 4$$

For $\lambda = \lambda_1 = 6$, we get

$$2x_1 - 4x_2 = 0$$
$$2x_1 - 4x_2 = 0$$

Therefore $x_1 = 2x_2$ and the corresponding eigenvector is

$$X_1 = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

Similarly, for $\lambda = \lambda_2 = 4$, we get $x_1 = x_2$ and the eigenvector is

$$X_2 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

The process of finding the eigenvalues and eigenvectors of large matrices is complex and involves a multistep procedure. There are several methods available and a discussion on all these methods will be beyond the scope of this book. We consider, in the next two sections, the following two methods:

1. Polynomial method
2. Power method

## 14.5 POLYNOMIAL METHOD

The polynomial method consists of the following three steps:

1. Determine the coefficients $p_i$ of the characteristic polynomial using the Fadeev-Leverrier method
2. Evaluate the roots (eigenvalues) of the characteristic polynomial using any of the root-finding techniques

3. Calculate the eigenvectors using any of the reduction techniques such as Gauss elimination

## The Fadeev-Leverrier Method

The Fadeev-Leverrier method evaluate the coefficients $p_i$, $i = 1, 2, ..., n$, of the characteristic polynomial

$$\lambda^n - p_1\lambda^{n-1} - p_2\lambda^{n-2} - .... - p_n = 0$$

The method consists of generating a sequence of matrices $A_i$ that can be employed to determine the $p_i$ values. The process is as follows:

$$A_1 = \mathbf{A} \tag{14.11}$$

$$p_1 = t_r A_1$$

Remaining values ($i = 2, 3, ..., n$) are evaluated from the recursive equations:

$$A_i = \mathbf{A}(A_{i-1} - p_{i-1}I)$$

$$p_i = \frac{t_r A_i}{i} \tag{14.12}$$

where $t_r A_i$ is the *trace* of the matrix $A_i$. Remember, the trace is the sum of the diagonal elements of the matrix.

### Example 14.4

Determine the coefficients of the characteristic polynomial of the system

$$(-1 - \lambda)x_1 = 0$$

$$x_1 + (-2 - \lambda)x_2 + 3x_3 = 0$$

$$2x_2 + (-3 - \lambda)x_3 = 0$$

using the Fadeev-Leverrier method.

The given system is a third-order one and, therefore, the characteristic polynomial takes the form

$$\lambda^3 - p_1\lambda^2 - p_2\lambda - p_3 = 0$$

The matrix A is given by

$$\mathbf{A} = \begin{bmatrix} -1 & 0 & 0 \\ 1 & -2 & 3 \\ 0 & 2 & -3 \end{bmatrix}$$

By using the Eq. (14.11),

$$A_1 = \mathbf{A}$$

$$p_1 = t_r A_1 = -6$$

By using the Eq. (14.12),

$$A_2 = \mathbf{A}(A_1 - p_1 I)$$

$$= \begin{bmatrix} -1 & 0 & 0 \\ 1 & -2 & 3 \\ 0 & 2 & -3 \end{bmatrix} \left\{ \begin{bmatrix} -1 & 0 & 0 \\ 1 & -2 & 3 \\ 0 & 2 & -3 \end{bmatrix} \right\} - \begin{bmatrix} -6 & 0 & 0 \\ 0 & -6 & 0 \\ 0 & 0 & -6 \end{bmatrix}$$

$$= \begin{bmatrix} -1 & 0 & 0 \\ 1 & -2 & 3 \\ 0 & 2 & -3 \end{bmatrix} \begin{bmatrix} 5 & 0 & 0 \\ 1 & 4 & 3 \\ 0 & 2 & 3 \end{bmatrix}$$

$$= \begin{bmatrix} -5 & 0 & 0 \\ -3 & -2 & 3 \\ 2 & 2 & -3 \end{bmatrix}$$

$$p_2 = \frac{t_r A_2}{2} = -5$$

Similarly,

$$A_3 = \mathbf{A}(A_2 - p_2 I)$$

$$= \begin{bmatrix} -1 & 0 & 0 \\ 1 & -2 & 3 \\ 0 & 2 & -3 \end{bmatrix} \left\{ \begin{bmatrix} -5 & 0 & 0 \\ -3 & -2 & 3 \\ 2 & 2 & -3 \end{bmatrix} \right\} - \begin{bmatrix} -5 & 0 & 0 \\ 0 & -5 & 0 \\ 0 & 0 & -5 \end{bmatrix}$$

$$= \begin{bmatrix} -1 & 0 & 0 \\ 1 & -2 & 3 \\ 0 & 2 & -3 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ -3 & 3 & 3 \\ 2 & 2 & 2 \end{bmatrix}$$

$$= \begin{bmatrix} 0 & 0 & 0 \\ 6 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$p_3 = \frac{t_r A_3}{3} = 0$$

Therefore, the characteristic polynomial is

$$\lambda^3 + 6\lambda^2 + 5\lambda = 0$$

or

$$\lambda(\lambda^2 + 6\lambda + 5) = 0$$

## Evaluating the Eigenvalues

Let us consider the characteristic polynomial obtained in Example 14.4.

$$\lambda(\lambda^2 + 6\lambda + 5) = 0$$

One of the roots is $\lambda_1 = 0$. The other two roots can be obtained using the familiar quadratic formula as

$$\lambda_2 = -1$$
$$\lambda_3 = -5$$

Since it is a quadratic equation, we can solve it using the quadratic formula. However, if the polynomial is of higher order, the roots may be evaluated using the techniques discussed in Chapter 6. We may use either the Newton-Raphson method with synthetic division or the Bairstow method.

Remember that the sum of the eigenvalues of a matrix is equal to the trace of that matrix. In the problem discussed above,

$$\text{trace of } \mathbf{A} = -1 - 2 - 3 = -6$$
$$\text{Sum of eigenvalues} = 0 - 1 - 5 = -6$$

## Determining the Eigenvectors

Once eigenvalues are evaluated, eigenvectors corresponding to these eigenvalues may be obtained by applying Gauss elimination method to the homogeneous equations. Example 14.5 illustrates this.

### Example 14.5

Determine the eigenvectors for the system discussed in Example 14.4.

The eigenvalues are:

$$\lambda_1 = 0, \qquad \lambda_2 = -1, \qquad \lambda_3 = -5$$

*Eigenvector 1* ($\lambda_1 = 0$)

The system of equations for $\lambda_1 = 0$ is

$$-x_1 = 0$$
$$x_1 - 2x_2 + 3x_3 = 0$$
$$2x_2 - 3x_3 = 0$$

This is equivalent to the system of two equations

$$x_1 = 0$$
$$2x_2 - 3x_3 = 0$$

Choosing $x_2 = 1,$

$$x_3 = 2/3 = 0.6667$$

*Eigenvector 2* ($\lambda_2 = -1$)

$$x_1 - x_2 + 3x_3 = 0$$
$$2x_2 - 2x_3 = 0$$

Choosing $x_2 = 1$, we get $x_3 = 1$ and $x_1 = -2$

*Eigenvector 3* ($\lambda_3 = -5$)

$$4x_1 = 0$$
$$x_1 + 3x_2 + 3x_3 = 0$$
$$2x_2 + 2x_3 = 0$$

$x_1 = 0$ and choosing $x_2 = 1$, we get $x_3 = -1$

The three eigenvectors are:

$$X_1 = \begin{bmatrix} 0 \\ 1 \\ 0.6667 \end{bmatrix}$$

$$X_2 = \begin{bmatrix} -2 \\ 1 \\ 1 \end{bmatrix}$$

$$X_3 = \begin{bmatrix} 0 \\ 1 \\ -1 \end{bmatrix}$$

## Computing Algorithm

The computing algorithm for polynomial method combines three different algorithms discussed so far. Algorithm 14.3 lists the major steps involved in implementing the polynomial method of finding eigenvalues and associated eigenvectors.

### Polynomial Method

1. Input order of the matrix and the elements of the matrix.
2. Determine the coefficients of the characteristic polynomial by using the Fadeev-Leverrier method.
3. Evaluate the roots of the polynomial using the Bairstow method (use Algorithm 6.10).
4. For each eigenvalue, construct the system of equations and then solve for the eigenvector using the Gauss elimination method (use Algorithm 7.2).
5. Print eigenvalues and the associated eigenvectors.

Algorithm 14.3

## 14.6 POWER METHOD

*Power method* is a 'single value' method used for determining the 'dominant' eigenvalue of a matrix. It is an iterative method implemented using an initial starting vector $X$. The starting vector can be arbitrary if no suitable approximation is available. Power method is implemented as follows:

$$Y = A\,X \tag{14.13}$$

$$X = \frac{1}{k} Y \tag{14.14}$$

The new value of $X$ obtained from Eq. (14.14) is used in Eq. (14.13) to compute a new value of $Y$ and the process is repeated until the desired level of accuracy is obtained. The parameter $k$, known as the *scaling factor*, is the element of $Y$ with the largest magnitude.

Let us assume that the eigenvalues are $|\lambda_1| > |\lambda_2| \geq \ldots \geq |\lambda_n|$ and the corresponding eigenvectors are $X_1, X_2, \ldots, X_n$. After repeated applications of Eqs (14.13) and (14.14), the vector $X$ converges to $X_1$ and $k$ converges to $\lambda_1$.

### Example 14.6

Find the largest eigenvalue $\lambda_1$ and the corresponding eigenvector $V_1$ of the matrix

$$\begin{bmatrix} 1 & 2 & 0 \\ 2 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

using the power method.

Let us assume the starting vector as

$$X = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

Equations (14.13) and (14.14) are repeatedly used as follows

*Iteration 1*

$$Y = \begin{bmatrix} 1 & 2 & 0 \\ 2 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix}$$

$$X = \frac{1}{2} \begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0.5 \\ 0 \end{bmatrix}$$

*Iteration 2*

$$Y = \begin{bmatrix} 1 & 2 & 0 \\ 2 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0.5 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ 2.5 \\ 0 \end{bmatrix}$$

$$X = \frac{1}{2.5} \begin{bmatrix} 2 \\ 2.5 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.8 \\ 1 \\ 0 \end{bmatrix}$$

The process is continued and the results are tabulated below:

| *Iteration* | | *1* | *2* | *3* | *4* | *5* | *6* | *7* |
|---|---|---|---|---|---|---|---|---|
| $Y$ | | 2.0 | 2.0 | 2.8 | 2.86 | 2.98 | 2.98 | 3.0 |
| | | 1.0 | 2.5 | 2.6 | 2.93 | 2.96 | 2.99 | 3.0 |
| | | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| $X$ | 0 | 1.0 | 0.8 | 1.0 | 0.98 | 1.0 | 1.0 | 1.0 |
| | 1 | 0.5 | 1.0 | 0.93 | 1.0 | 0.99 | 1.0 | 1.0 |
| | 0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |

The final entry in the table shows that $\lambda_1 = 3.0$ (element of $\boldsymbol{Y}$ with largest magnitude) and the corresponding eigenvector is the last $\boldsymbol{X}$. That is,

$$\boldsymbol{X}_1 = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}$$

Algorithm 14.4 gives an implementation of the power method. Note that the stopping criterion is based on the successive values of the vector $\boldsymbol{X}$. There might be circumstances where the process will not converge at all (or where it will converge very slowly). Therefore, it is necessary to put a limitation on the number of iterations.

**Power Method**

```
1. Input matrix A, initial vector X, error tolerance (EPS) and maximum
   iterations permitted (MAXIT).
2. Compute $Y = AX$
3. Find the element $k$ of $Y$ that is largest in magnitude.
4. Compute $X = Y/k$
5. If $|X - X_{oid}|$ < EPS or Iterations > MAXIT
     write $k$ and $X$
   else
     go to step 2
```

**Algorithm 14.4**

Engineers often come across cases where they are interested in the smallest eigenvalue of the system. The smallest eigenvalue can be determined by applying the power method to the matrix inverse of [A].

## 14.7 SUMMARY

We considered two classes of problems in this chapter:

- boundary value problems
- eigenvalue problems

We presented two methods for solving boundary-value problems:

- shooting method
- finite difference method

We also discussed in detail the nature and solution of eigenvalue problems and presented two methods for evaluating eigenvalues and eigenvectors:

- power method
- polynomial method

### Key Terms

*Boundary value problem*
*Characteristic equation*
*Characteristic matrix*
*Characteristic polynomial*
*Characteristic values*
*Eigenvalue problem*
*Eigenvalues*
*Eigenvector*
*Fadeev-Leverrier method*
*Finite difference method*
*Identity matrix*
*Initial-value problem*
*Polynomial method*
*Power method*
*Scaling factor*
*Shooting method*
*Trace of the matrix*

## REVIEW QUESTIONS

1. What is a boundary-value problem? How is it different from an initial-value problem?
2. State the two popular methods used for solving boundary-value problems.
3. What are eigenvalue problems? How are they different from boundary-value problems?
4. State at least two methods used for solving eigenvalue problems.
5. Describe the shooting method with graphical illustration.
6. Explain the concept employed in the finite difference method.
7. Define the following:
   (a) Identity matrix
   (b) Characteristic matrix
   (c) Characteristic polynomial or equation
   (d) Eigenvalues
   (e) Eigenvectors
   (f) Trace of a matrix
8. What is Fadeev-Leverrier method used for? Explain.
9. Describe the algorithm of polynomial method used for solving eigenvalue problems.
10. Describe the implementation of power method with the help of a flow chart.

## REVIEW EXERCISES

1. Solve the following equations using the shooting method.

   (a) $\dfrac{d^2y}{dx^2} = 6x + 4, \qquad y(0) = 2 \quad \text{and} \quad y(1) = 5$

   (*b*) $\dfrac{d^2y}{dx^2} = 12x^2, \qquad y(1) = 2 \quad \text{and} \quad y(2) = 17$

2. Use the finite difference approach to solve the equations in Exercise 1 with $\Delta x = 0.2$.
3. Use the shooting method to solve the following differential equation:

$$\frac{d^2y}{dx^2} + 2\frac{dy}{dx} - \frac{y}{2} - 2.5 = 0$$

   given the boundary conditions $y(0) = 10$ and $y(10) = 6$.

4. Solve Exercise 3 using the finite difference method with $\Delta x = 2$.
5. Given the boundary-value problem

$$\frac{d^2y}{dx^2} = 3x + 4y, \qquad y(0) = 1 \qquad \text{and} \qquad y(1) = 1$$

   obtain its solution in the range $0 \leq x \leq 1$ with $\Delta x = 0.25$ using
   (a) shooting method
   (b) finite difference method

6. Given the equation

$$x^2\frac{d^2y}{dx^2} - 2x\frac{dy}{dx} - 2y + x^2 \sin x = 0$$

   and boundary conditions $y(1) = 1$ and $y(2) = 2$, estimate $y(1.25)$, $y(1.5)$ and $y(1.75)$ using the shooting method.

7. Solve the following boundary-value problems using a suitable method.

   (a) $\dfrac{d^2y}{dx^2} - 6y^2 = 0, \qquad y(1) = 1 \quad \text{and} \quad y(2) = 0.25$

   (b) $\dfrac{d^2y}{dx^2} + e^{-2y} = 0, \qquad y(1) = 0 \quad \text{and} \quad y(2) = 1$

   (c) $\dfrac{d^2y}{dx^2} - 3\dfrac{dy}{dx} + 2y = 2, \qquad y(0) = 1 \quad \text{and} \quad y(1) = 4$

   (d) $\dfrac{d^2y}{dx^2} - x\dfrac{dy}{dx} + y = -x^2, \qquad y(0) = -2 \quad \text{and} \quad y(1) = 1$

8. Find the characteristic polynomials of the following systems using Fadeev-Leverrier method.

   (a) $2x_1 + 8x_2 + 10x_3 = \lambda x_1$

$$8x_1 + 3x_2 + 4x_8 = \lambda x_2$$
$$10x_1 + 4x_2 + 7x_3 = \lambda x_3$$

(b) $16x_1 - 24x_2 + 18x_3 = \lambda x_1$
$3x_1 - 2x_2 = \lambda x_2$
$-9x_1 + 18x_2 - 17x_3 = \lambda x_3$

(c) $2x_1 + 2x_2 + 2x_3 = \lambda x_1$
$2x_1 + 5x_2 + 5x_3 = \lambda x_2$
$2x_1 + 5x_2 + 1x_3 = \lambda x_3$

9. Evaluate the eigenvectors of the systems given in Exercise 8.
10. Find the largest eigenvalue and the corresponding eigenvector of the following matrices using the power method.

(a) $A = \begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{bmatrix}$

(b) $B = \begin{bmatrix} -1 & 0 & 0 \\ 1 & -2 & 3 \\ 0 & 2 & -3 \end{bmatrix}$

(c) $C = \begin{bmatrix} -13 & 3 & -5 \\ 0 & -4 & 0 \\ 15 & -9 & 7 \end{bmatrix}$

## PROGRAMMING PROJECTS

1. Develop a program to implement the shooting method algorithm for a linear, second-order ordinary differential equation.
2. Develop a program to implement the finite difference method of solving a linear, second-order ordinary differential equation.
3. Develop a modular program which uses the following subprograms to implement the polynomial method.
   (a) Subprogam to obtain the coefficients of the characteristic polynomial using Fadeev-Leverrier method.
   (b) Subprogram to evaluate the roots of the characteristic polynomial (use Bairstow method).
   (c) Subprogram to determine the eigenvectors (use Gauss elimination method).

   You may also use subprograms to receive input information and print output information.
4. Develop a user-friendly program to evaluate the largest eigenvalue and the corresponding eigenvector using the power method.
5. Develop a program to compute the smallest eigenvalue using the power method.

CHAPTER 15

# Solution of Partial Differential Equations

## 15.1 NEED AND SCOPE

Many physical phenomena in applied science and engineering when formulated into mathematical models fall into a category of systems known as *partial differential equations*. A partial differential equation is a differential equation involving more than one independent variable. These variables determine the behaviour of the dependent variable as described by their *partial derivatives* contained in the equation. Some of the problems which lend themselves to partial differential equations include:

1. Study of displacement of a vibrating string,
2. Heat flow problems,
3. Fluid flow analysis,
4. Flectrical potential distribution,
5. Analysis of torsion in a bar subject to twisting,
6. Study of diffusion of matter, and so on.

Most of these problems can be formulated as second-order partial differential equations (with the highest order of derivative being the second). If we represent the dependent variable as $f$ and the two independent variables as $x$ and $y$, then we will have three possible second-order partial derivatives $\frac{\partial^2 f}{\partial x^2}$, $\frac{\partial^2 f}{\partial x\, \partial y}$ and $\frac{\partial^2 f}{\partial y^2}$ in addition to the two first-order partial derivatives $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$.

We can write a second-order equation involving two independent variables in general form as

$$a\frac{\partial^2 f}{\partial x^2}+b\frac{\partial^2 f}{\partial x\,\partial y}+c\frac{\partial^2 f}{\partial y^2}=F\left(x, y, f, \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right) \tag{15.1}$$

where the coefficients $a$, $b$, and $c$ may be constants or functions of $x$ and $y$. Depending on the values of these coefficients, Eq. (15.1) may be classified into one of the three types of equations, namely, *elliptic, parabolic*, and *hyperbolic*.

| | |
|---|---|
| Elliptic, | if $b^2 - 4ac < 0$ |
| Parabolic, | if $b^2 - 4ac = 0$ |
| Hyperbolic, | if $b^2 - 4ac > 0$ |

If $a$, $b$, and $c$ are functions of $x$ and $y$, then depending on the values of these coefficients at various points in the domain under consideration, an equation may change from one classification to another.

Solution of partial differential equations is too important to ignore but too difficult to cover in depth in an introductory book. Since the application of analytical methods becomes more complex, we seek the help of numerical techniques to solve partial differential equations. There are basically two numerical techniques, namely, *finite-difference* method and *finite-element* method that can be used to solve partial differential equations (PDEs). Although the finite-element method is very important for solving equations where regions are irregular, discussion on this technique is beyond the scope of this book. We will discuss here the application of finite-difference methods only, which are based on formulae for approximating the first and second derivatives of a function. We will also consider problems, only those where the coefficients $a$, $b$, and $c$ are constants.

## 15.2 DERIVING DIFFERENCE EQUATIONS

In this section, we will discuss two-dimensional problems only. Consider a two-dimensional solution domain as shown in Fig. 15.1. The domain is split into regular rectangular grids of width $h$ and height $k$. The pivotal values at the points of intersection (known as grid points or nodes) are denoted by $f_{ij}$ which is a function of the two space variables $x$ and $y$.

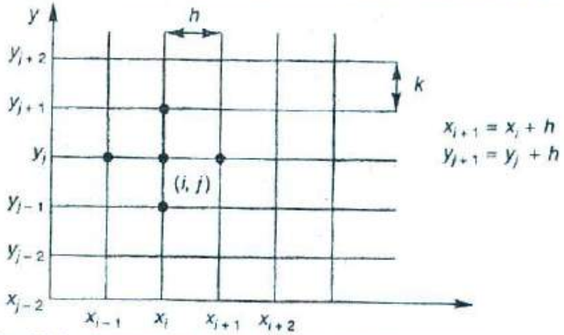


Fig. 15.1 Two-dimensional finite difference grid

In the finite difference method, we replace derivatives that occur in the PDE by their finite difference equivalents. We then write the difference equation corresponding ‘ each "grid point" (where derivative is required) using function values at the surrounding grid points. Solving these equations simultaneously gives values for the function at each grid point.

We have a. ssed in Chapter 11 that, if the function $f(x)$ has a continuous four.n derivative, then its first and second derivatives are given by the following central difference approximations.

$$f'(x_i) = \frac{f(x_i + h) - f(x_i - h)}{2h}$$

or

$$f'_i = \frac{f_{i+1} - f_{i-1}}{2h} \tag{15.2}$$

$$f''(x_i) = \frac{f(x_i + h) - 2f(x_i) + f(x_i - h)}{h^2}$$

or

$$f''_i = \frac{f_{i+1} - 2f_i + f_{i-1}}{h^2} \tag{15.3}$$

The subscript on $f$ indicates the $x$-value at which the function is evaluated.

When $f$ is a function of two variables $x$ and $y$, the partial derivatives of $f$ with respect $x$ (or $y$) are the ordinary derivatives of $f$ with respect to $x$ (or $y$) when $y$ (or $x$) does not change. We can use Eqs (15.2) and (15.3) in the $x$-direction to determine derivatives with respect to $x$ and in the $y$-direction to determine derivatives with respect to $y$. Thus, we have

$$\frac{\partial f(x_i, y_j)}{\partial x} = f_x(x_i, y_j) = \frac{f(x_{i+1}, y_j) - f(x_{i-1}, y_j)}{2h}$$

$$\frac{\partial f(x_i, y_j)}{\partial y} = f_y(x_i, y_j) = \frac{f(x_i, y_{j+1}) - f(x_i, y_{j-1})}{2k}$$

$$\frac{\partial^2 f(x_i, y_j)}{\partial x^2} = f_{xx}(x_i, y_j) = \frac{f(x_{i+1}, y_j) - 2f(x_i, y_j) + f(x_{i-1}, y_j)}{h^2}$$

$$\frac{\partial^2 f(x_i, y_j)}{\partial y^2} = f_{yy}(x_i, y_j) = \frac{f(x_i, y_{j+1}) - 2f(x_i, y_j) + f(x_i, y_{j-1})}{k^2}$$

$$\frac{\partial^2 f(x_i, y_j)}{\partial x\, \partial y} = \frac{f(x_{i+1}, y_{j+1}) - f(x_{i+1}, y_{j-1}) - f(x_{i-1}, y_{j+1}) + f(x_{i-1}, y_{j-1})}{4hk}$$

It is convenient to use double subscripts $i, j$ on $f$ to indicate $x$ and $y$ values. Then, the above equations become

$$f_{x,\,ij} = \frac{f_{i+1,j} - f_{i-1,j}}{2h} \tag{15.4}$$

$$f_{y,\,ij} = \frac{f_{i,j+1} - f_{i,j-1}}{2k} \tag{15.5}$$

$$f_{xx,\,ij} = \frac{f_{i+1,j} - 2f_{i,j} + f_{i-1,j}}{h^2} \tag{15.6}$$

$$f_{yy,\,ij} = \frac{f_{i,j+1} - 2f_{ij} + f_{i,j-1}}{k^2} \tag{15.7}$$

$$f_{xy,\,ij} = \frac{f_{i+1,j+1} - f_{i+1,j-1} - f_{i-1,j+1} + f_{i-1,j-1}}{4hk} \tag{15.8}$$

We will use these finite difference equivalents of the partial derivatives to construct various types of differential equations.

## ELLIPTIC EQUATIONS

Elliptic equations are governed by conditions on the boundary of closed domain. We consider here the two most commonly encountered elliptic equations, namely,

Laplace's equation, and Poisson's equation.

### Laplace's Equation

Equation (15.1), when $a = 1, b = 0, c = 1$, and $F(x, y, f, f_x, f_y) = 0$, becomes

$$\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} = \nabla^2 f = 0 \tag{15.9}$$

The operator

$$\nabla^2 = \left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right)$$

is called the *Laplacian operator* and Eq. (15.9) is called *Laplace's equation*. (Many authors use $u$ in place of $f$.)

To solve the Laplace equation on a region in the $xy$-plane, we subdivide the region as shown in Fig. 15.1. Consider the portion of the region near $(x_i, y_i)$. We have to approximate

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} = 0$$

Replacing the second-order derivatives by their finite difference equivalents (from Eqs (15.6) and (15.7)) at the point $(x_i, y_i)$, we obtain,

$$\nabla^2 f_{ij} = \frac{f_{i+1,j} - 2f_{ij} + f_{i-1,j}}{h^2} + \frac{f_{i,j+1} - 2f_{ij} + f_{i,j-1}}{k^2}$$

If we assume, [illegible] implicity, $h = k$, then we get

$$\nabla^2 f_{ij} = \frac{1}{h^2}\,(f_{i+1,j} + f_{i-1,j} - 4f_{ij} + f_{i,j+1} + f_{i,j-1}) = 0 \qquad (15.10)$$

Note that Eq. (15.10) contains four neighbouring points around the central point $(x_i, y_j)$ (on all the four sides) as shown in Fig. 15.2. Equation (15.10) is known as the *five-point difference formula* for Laplace's equation.

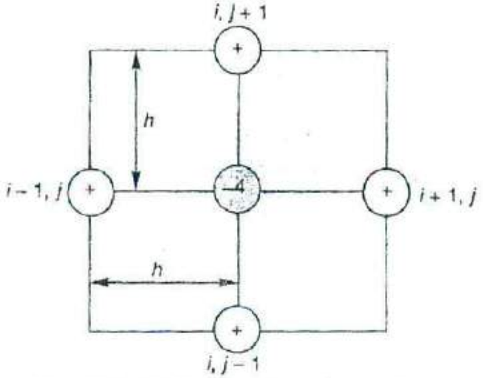


**Fig. 15.2** Grid for Laplace's equation

We can also represent the relationship of pivotal values pictorially as in Eq. (15.11).

$$\nabla^2 f_{ij} = \frac{1}{h^2}\left\{\begin{matrix} & 1 & \\ 1 & -4 & 1 \\ & 1 & \end{matrix}\right\} f_{ij} = 0 \qquad (15.11)$$

From Eq. (15.10) we can show that the function value at the grid point $(x_i, y_j)$ is the average of the values at the four adjoining points. That is,

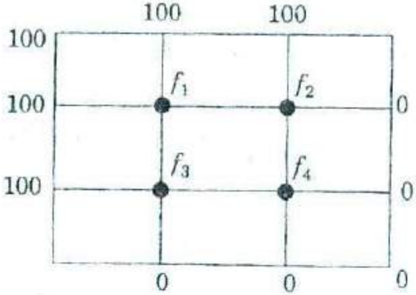
$$\boxed{f_{ij} = \frac{1}{4}(f_{i+1,j} + f_{i-1,j} + f_{i,j+1} + f_{i,j-1})} \qquad (15.12)$$

To evaluate numerically the solution of Laplace's equation at the grid points, we can apply Eq. (15.12) at the grid points where $f_{ij}$ is required (or unknown), thus obtaining a system of linear equations in the pivotal values $f_{ij}$. The system of linear equations may be solved using either direct methods or iterative methods.

**Example 15.1**

Consider a steel plate of size 15 cm × 15 cm. If two of the sides are held at 100°C and the other two sides are held at 0°C, what are the steady-state temperature at interior points assuming a grid size of 5 cm × 5 cm.

A problem with the values known on each boundary is said to have *Dirichlet boundary conditions*. The problem is illustrated below.



The system of equations is as follows:

At point 1: $f_2 + f_3 - 4f_1 + 100 + 100 = 0$

At point 2: $f_1 + f_4 - 4f_2 + 100 + 0 = 0$

At point 3: $f_1 + f_4 - 4f_3 + 100 + 0 = 0$

At point 4: $f_2 + f_3 - 4f_4 + 0 + 0 = 0$

That is,

$$\begin{aligned} -4f_1 + f_2 + f_3 + 0 &= -200 \\ f_1 + -4f_2 + 0 + f_4 &= -100 \\ f_1 + 0 - 4f_3 + f_4 &= -100 \\ 0 + f_2 + f_3 - 4f_4 &= 0 \end{aligned}$$

Solution of this system are

$$f_1 = 75 \qquad f_2 = 50$$
$$f_3 = 50 \qquad f_4 = 25$$

Note that there is a symmetry in the temperature distribution, i.e. it can be stated that

$$f_2 = f_3$$

and therefore the number of equations in Example 15.1 may be reduced to three with three unknowns as shown below.

$$\begin{aligned} -4f_1 + 2f_2 &= -200 \\ f_1 - 4f_2 + f_4 &= -100 \\ f_2 - 2f_4 &= 0 \end{aligned} \qquad (15.13)$$

## Liebmann's Iterative Method

We know that a diagonally domina~~t system of linear equations can be solved by iteration methods such a. Gauss-Seidel method. When such an iteration is applied to Laplace's equation, the iterative method is called *Liebmann's iterative method.*

To obtain t. ivotal values of $f$ by Liebmann's iterative method, we solve for $f_{ij}$ the e,uations obtained from Eq. (15.10). That is,

$$\boxed{f_{ij} = \frac{1}{4}(f_{i+1,j} + f_{i-1,j} + f_{i,j+1} + f_{i,j-1})} \tag{15.14}$$

The value $f_{ij}$ at the point $ij$ is the average of the values of $f$ at the four adjoining points. If we know the "initial values" of the functions at the right-hand side of Eq. (15.13), we can estimate the value $f$ at the point $ij$. We can substitute the values thus obtained into the right-hand side to achieve improved approximations. This process may continue till the values $f_{ij}$ converge to constant values.

Initial values may be obtained by either taking *diagonal average* or *cross average* of the adjoining four points.

### Example 15.2

Solve the problem in Examole 15.1 using Liebmann's iterative method correct to one decimal place.

By applying Eq. (15.14) to every grid point, we obtain

$$\begin{aligned}
f_1 &= \frac{f_2 + f_3 + 200}{4} \\
f_2 &= \frac{f_1 + f_4 + 100}{4} \\
f_3 &= \frac{f_1 + f_4 + 100}{4} \\
f_4 &= \frac{f_2 + f_3}{4}
\end{aligned} \tag{15.15}$$

Appropriate initial values for the iterative solution are obtained by taking diagonal average at 1 and cross average at other points, assuming first $f_4 = 0$.

$$f_1 = \frac{1}{4}(100 + 100 + 100 + 0) = 75.00 \text{ (average)}$$

$$f_2 = \frac{1}{4}(75 + 100 + 0 + 0) = 43.75$$

$$f_3 = \frac{1}{4}(100 + 75 + 0 + 0) = 43.75$$

$$f_4 = \frac{1}{4}(43.75 + 43.75 + 0 + 0) = 21.88$$

Note that $f_2$, $f_3$, and $f_4$ are computed using the latest values on the right-hand side.

Using these initial values in Eq. (15.15) and performing iterations gives the values as shown in Table 15.1.

Table 15.1

| $f_i$ | *Initial Values* | *Iterations* | | | |
|---|---|---|---|---|---|
| | | *1* | *2* | *3* | *4* |
| $f_1$ | 75.00 | 71.88 | 74.22 | 74.81 | 74.95 |
| $f_2$ | 43.75 | 48.44 | 49.61 | 40.90 | 49.98 |
| $f_3$ | 43.75 | 48.44 | 49.61 | 49.90 | 49.98 |
| $f_4$ | 21.88 | 24.22 | 24.81 | 24.95 | 24.99 |

The process may be continued till we get identical values in the last two columns. Note that the values are approaching to correct answers obtained in Example 15.1.

## Poisson's Equation

Equation (15.1), when $a = 1$, $b = 0$, $c = 1$ and $F(x, y, f, f_x, f_y) = g(x, y)$, becomes

$$\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} = g(x, y) \tag{15.16}$$

or

$$\nabla^2 f = g(x, y)$$

Equation (15.16) is called *Poisson's equation*. Using the notation $g_{ij} = g(x_i, y_j)$, Eq. 15.10 used for Laplace's equation may be modified to solve Eq. 15.16. The finite difference formula for solving Poisson's equation then takes the form

$$\boxed{f_{i+1,j} + f_{i-1,j} + f_{i,j+1} + f_{i,j-1} - 4f_{ij} = h^2 g_{ij}} \tag{15.17}$$

By applying the replacement formula to each grid point in the domain of consideration, we will get a system of linear equations in terms of $f_{ij}$. These equations may be solved either by any of the elimination methods or by any iteration techniques as done in solving Laplace's equation.

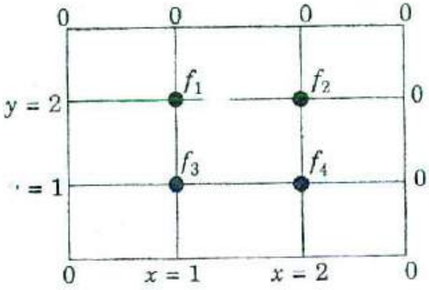
### Example 15.3

Solve the Poisson equation

$$\nabla^2 f = 2x^2y^2$$

over the square domain $0 \le x \le 3$ and $0 \le y \le 3$ with $f = 0$ on the boundary and $h = 1$.

The domain is divided into squares of one unit size as illustrated below:



By applying Eq. (15.17) at each grid point, we get the following set of equations:

Point 1: $0 + 0 + f_2 + f_3 - 4f_1 = 2(1)^2(2)^2$

$$\text{i.e.} \quad f_2 + f_3 - 4f_1 = 8 \tag{a}$$

Point 2: $0 + 0 + f_1 + f_4 - 4f_2 = 2(2)^2(2)^2$

$$\text{i.e.} \quad f_1 - 4f_2 + f_4 = 32 \tag{b}$$

Point 3: $0 + 0 + f_1 + f_4 - 4f_3 = 2(1)^2(1)^2$

$$\text{i.e.} \quad f_1 - 4f_3 + f_4 = 2 \tag{c}$$

Point 4: $0 + 0 + f_2 + f_3 - 4f_4 = 2(2)^2(1)^2$

$$\text{i.e.} \quad f_2 + f_3 - 4f_4 = 8 \tag{d}$$

Rearranging the equations (a) to (d), we get

$$\begin{aligned} -4f_1 + f_2 + f_3 \quad &= \ 8 \\ f_1 - 4f_2 \quad + f_4 &= 32 \\ f_1 \quad - 4f_3 + f_4 &= \ 2 \\ f_2 + f_3 - 4f_4 &= \ 8 \end{aligned}$$

Solving these equations by elimination method, we get the answers.

$$f_1 = -\frac{22}{4}, \qquad f_2 = -\frac{43}{4}$$

$$f_3 = -\frac{13}{4}, \qquad f_4 = -\frac{22}{4}$$

## Example 15.4

Solve the problem in Example 15.3 by Gauss-Seidel iteration method.

By rearranging the equations used in Example 15.3, we have

$$f_1 = \frac{1}{4}(f_2 + f_3 - 8)$$

$$f_2 = \frac{1}{4}(f_1 + f_4 - 32)$$

$$f_3 = \frac{1}{4}(f_1 + f_4 - 2)$$

$$f_4 = \frac{1}{4}(f_2 + f_3 - 8)$$

Note that $f_1 = f_4$
Therefore,

$$f_1 = \frac{1}{4}(f_2 + f_3 - 8)$$

$$f_2 = \frac{1}{4}(2f_1 - 32)$$

$$f_3 = \frac{1}{4}(2f_1 - 2)$$

Assume starting values as $f_2 = 0 = f_3$

*Iteration 1*

$$f_1 = -2, \quad f_2 = -9, \quad f_3 = -1$$

*Iteration 2*

$$f_1 = -\frac{18}{4}, \quad f_2 = -\frac{41}{4}, \quad f_3 = -\frac{11}{4}$$

*Iteration 3*

$$f_1 = -\frac{22}{4}, \quad f_2 = -\frac{43}{4}, \quad f_3 = -\frac{13}{4}$$

*Iteration 4*

$$f_1 = -\frac{22}{4}, \quad f_2 = -\frac{43}{4}, \quad f_3 = -\frac{13}{4}$$

## 15.4 PARABOLIC EQUATIONS

Elliptic equations studied previously describe problems that are time-independent. Such problems are known as steady-state problems. But we come across problems that are not steady-state. This means that the function is dependent on both space and time. Parabolic equations, for which

$$b^2 - 4ac = 0$$

describe the problems that depend on space and time variables.

A popular case for parabolic type of equation is the study of heat flow in one-dimensional direction in an insulated rod. Such problems are governed by both boundary and initial conditions.

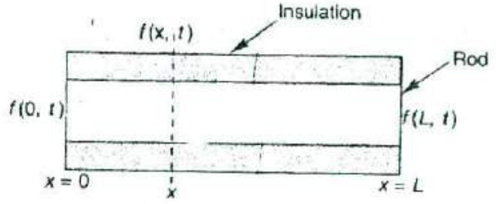


**Fig. 15.3** Heat flow in a rod

Let $f$ represent the temperature at any point in rod (Fig. 15.3) whose distance from the left end is $x$. Heat is flowing from left to right under the influence of temperature gradient. The temperature $f(x, t)$ in the rod at the position $x$ and time $t$, is governed by the *heat equation*

$$k_1 \frac{\partial^2 f}{\partial x^2} = k_2 k_3 \frac{\partial f}{\partial t} \tag{15.18}$$

where $k_1$ = Coefficient of thermal conductivity; $k_2$ = Specific heat; and $k_3$ = Density of the material.

Equation (15.18) may be simplified as

$$\boxed{k f_{xx}(x, t) = f_t(x, t)} \tag{15.19}$$

where

$$k = \frac{k_1}{k_2 k_3}$$

The initial condition will be the initial temperatures at all points along the rod.

$$f(x, 0) = f(x), \qquad 0 \le x \le L$$

The boundary conditions $f(0, t)$ and $f(L, t)$ describe the temperature at each end of the rod as functions of time. If they are held constant, then

$$f(0, t) = c_1, \qquad 0 \le t < \infty$$

$$f(L, t) = c_2, \qquad 0 \le t < \infty$$

## Solution of Heat Equation

We can solve the heat equation given by Eq. (15.19) using the finite difference formulae given below:

$$f_t(x, t) = \frac{f(x, t+\tau) - f(x, t)}{\tau}$$

$$= \frac{1}{\tau}(f_{i, j+1} - f_{i, j}) \tag{15.20}$$

$$f_{xx}(x, t) = \frac{f(x-h, t) - 2f(x, t) + f(x+h, t)}{h^2}$$

$$= \frac{1}{h^2}(f_{i-1,j} - 2f_{i,j} + f_{i+1,j}) \tag{15.21}$$

Substituting (15.20) and (15.21) in (15.19), we obtain

$$\frac{1}{\tau}(f_{i,j+1} - f_{i,j}) = \frac{k}{h^2}(f_{i-1,j} - 2f_{i,j} + 2f_{i}+_{1,j}) \tag{15.22}$$

Solving for $f_{i,j+1}$

$$f_{i,j+1} = \left(1 - \frac{2\tau k}{h^2}\right) f_{ij} + \frac{\tau k}{h^2}(f_{i-1,j} + f_{i+1,j})$$

$$= (1 - 2r)f_{ij} + r(f_{i-1,j} + f_{i+1,j}) \tag{15.23}$$

where $$r = \frac{\tau k}{h^2}$$

## Bender-Schmidt Method

The recurrence Eq. (15.23) allows us to evaluate $f$ at each point $x$ and at any time $t$. If we choose step sizes $\Delta t$ and $\Delta x$ such that

$$1 - 2r = 1 - \frac{2\tau k}{h^2} = 0 \tag{15.24}$$

Equation 15.23 simplifies to

$$\boxed{f_{i,j+1} = \frac{1}{2}(f_{i+1,j} + f_{i-1,j})} \tag{15.25}$$

Equation 15.25 is known as the *Bender-Schmidt recurrence equation*. This equation determines the value of $f$ at $x = x_i$, at time $t = t_j + \tau$, as the average of the values right and left of $x_i$ at time $t_j$.

Note that the step size in time $\Delta t$ obtained from Eq. (15.24)

$$\tau = \frac{h^2}{2k}$$

gives the Eq. (15.25). Equation (15.23) is stable, if and only if the step size $\tau$ satisfies the condition $\tau \le \frac{h^2}{2k}$.

### Example 15.5

Solve the equation

$$2f_{xx}(x, t) = f_t(x, t), \qquad 0 < t < 1.5 \qquad \text{and} \qquad 0 < x < 4$$

given the initial condition

$$f(x, 0) = 50(4 - x), \qquad 0 \le x \le 4$$

and the boundary conditions

$$f(0, t) = 0, \qquad 0 \le t \le 1.5$$
$$f(4, t) = 0, \qquad 0 \le t \le 1.5$$

If we assume $\Delta x = h = 1$, $\Delta t = \tau$ must be

$$\tau \le \frac{1^2}{2 \times 2} = 0.25$$

Taking $\tau = 0.25$, we have

$$f_{i,j+1} = \frac{1}{2}(f_{i-1,j} + f_{i+1,j})$$

Using this formula, we can generate successfully $f(x, t)$. The estimates are recorded in Table 15.2. At each interior point, the temperature at any single point is just average of the values at the adjacent points of the previous time value.

Table 15.2

| $t$ \ $x$ | *0.0* | *1.0* | *2.0* | *3.0* | *4.0* |
|---|---|---|---|---|---|
| 0.00 | 0.0 | 150.0 | 100.0 | 50.0 | 0.0 |
| 0.25 | 0.0 | 50.0 | 100.0 | 50.0 | 0.0 |
| 0.50 | 0.0 | 50.0 | 50.0 | 50.0 | 0.0 |
| 0.75 | 0.0 | 25.0 | 25.0 | 25.0 | 0.0 |
| 1.00 | 0.0 | 12.5 | 25.0 | 12.5 | 0.0 |
| 1.25 | 0.0 | 12.5 | 12.5 | 12.5 | 0.0 |
| 1.50 | 0.0 | 6.25 | 12.5 | 6.25 | 0.0 |

$f(x,0) = 50(4 - x)$

## The Crank-Nicholson Method

Solution of the parabolic equation given in Eq. (15.19) was solved using a forward difference formula in Eq. (15.20) for the time derivative and a central difference formula in Eq. (15.21) for the space derivative. This is called *explicit method* because all starting values are directly available from initial and boundary conditions and each new value is obtained from the values that are already known.

Accuracy of the explicit method may be improved if we use central difference formulae for both time and space derivatives. The forward difference quotient used for the time derivative (Eq. (15.20))

$$\frac{f_{i,j+1} - f_{i,j}}{\tau}$$

may be treated as central difference if we consider it to represent the *midpoint of the time interval* $(j, j + 1)$. We can also use the central difference quotient for the second derivative with distance, corresponding to the midpoint in time. Then,

$$f_t\left(x, t+\frac{\tau}{2}\right) = \frac{1}{\tau}(f_{i,j+1} - f_{i,j}) \tag{15.26}$$

The central difference quotient for the second-order space derivative is obtained by taking the average of difference quotients at the beginning and end of the time step, i.e.

Central difference at $t_{j+1/2} = \frac{1}{2}$ (Central difference at $t_j$ + Central difference at $t_{j+1}$).

Then,

$$k\,f_{xx}(x, t+\tau/2) = \frac{k}{2}\left(\frac{f_{i-1,j} + f_{i+1,j} - 2f_{i,j}}{h^2} + \frac{f_{i-1,j+1} + f_{i+1,j+1} - 2f_{i,j+1}}{h^2}\right) \tag{15.27}$$

Then equating Eqs (15.26) and (15.27) and substituting

$$r = \frac{\tau k}{h^2}$$

We get

$$\boxed{-rf_{i-1,j+1} + (2+2r)f_{i,j+1} - rf_{i,j+1} = rf_{i-1,j} + (2-2r)f_{i,j} + rf_{i+1,j}} \tag{15.28}$$

Equation (15.28) is called the *Crank-Nicholson formula*. If we let $r = 1$, then Eq (15.28) simplifies to

$$\boxed{-f_{i-1,j+1} + 4f_{i,j+1} - f_{i+1,j+1} = f_{i-1,j} + f_{i+1,j}} \tag{15.29}$$

The terms on the right-hand side are all known. Hence Eq. (15.29) forms a system of linear equations. The points used in the Crank-Nicholson formula are shown in Fig. 15.4. The boundary conditions are used in the first and last equations, i.e.

$$f_{1,j} = f_{1,j+1} = c_1$$
$$f_{n,j} = f_{n,j+1} = c_2$$

The Crank-Nicholson formula is called an *implicit method* because the values to be computed are not just a function of values at the previous time step, but also involve the values at the same time step which are not readily available. This requires us to solve a set of simultaneous equations at each time step.

Referring to Fig. 15.4,

$$-f_A + 4f_B - f_C = f_D + f_F$$

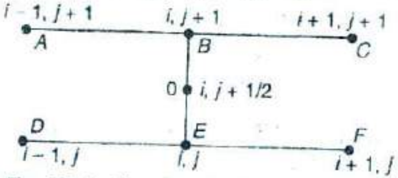


Fig. 15.4 The Crank-Nicholson grid

### Example 15.6

Solve the problem in Example 15.5 by the Crank-Nicholson implicit method.

Let us use a table as shown in Table 15.3 for recording the function values at various time steps. The values for the first time step ($t = 0$) are obtained from the initial condition

$$f(x, 0) = 50(4 - x)$$

and the values for $f_1$ and $f_5$ are obtained from the boundary conditions.

Table 15.3

| $t_j$ | $x = 0$<br>$f_1$ | $x = 1$<br>$f_2$ | $x = 2$<br>$f_3$ | $x = 3$<br>$f_4$ | $x = 4$<br>$f_5$ |
|---|---|---|---|---|---|
| $t_1 = 0.00$ | 0.0 | 150.00 | 100.00 | 50.00 | 0.0 |
| $t_2 = 0.25$ | 0.0 | 56.25 | 75.00 | 43.75 | 0.0 |
| $t_3 = 0.50$ | 0.0 | | | | 0.0 |
| $t_4 = 0.75$ | 0.0 | | | | 0.0 |
| $t_5 = 1.00$ | 0.0 | | | | 0.0 |

Now, for the second time step ($t = 0.25$), we write equations at each point using Eq. (15.29) and solve for unknowns. Thus, for the second row in the table, we have

$$-0.0 + 4f_2 - f_3 = 0.0 + 150$$
$$-f_2 + 4f_3 - f_4 = 150 + 150$$
$$-f_3 + 4f_4 - 0.0 = 100 + 0.0$$

Solving these three equations for three unknowns, we obtain

$$f_2 = 56.25$$
$$f_3 = 75.00$$
$$f_4 = 43.75$$

This process may be continued for each time step. Students may complete the table.

## 15.5 HYPERBOLIC EQUATIONS

Hyperbolic equations model the vibration of structures such as buildings, beams and machines. We consider here the case of a vibrating string that is fixed at both the ends as shown in Fig. 15.5.

The lateral displacement of string $f$ varies with time $t$ and distance $x$ along the string. The displacement $f(x, t)$ is governed by the *wave equation*

$$T\frac{\partial^2 f}{\partial x^2} = \rho\frac{\partial^2 f}{\partial t^2}$$

where $T$ is the tension in the string and $\rho$ is the mass per unit length.

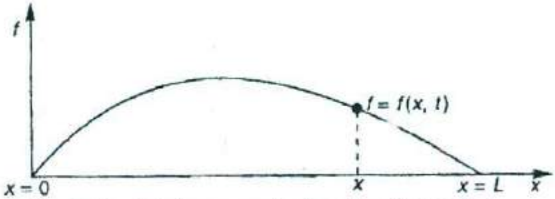


Fig. 15.5 Displacement of a vibrating string

Hyperbolic problems are also governed by both boundary and initial conditions, if time is one of the independent variables. Two boundary conditions for the vibrating string problem under consideration are

$$f(0, t) = 0 \qquad 0 \le t \le b$$

$$f(L, t) = 0 \qquad 0 \le t \le b$$

Two initial conditions are

$$f(x, 0) = f(x) \qquad 0 \le x \le a$$

$$f_t(x, 0) = g(x) \qquad 0 \le x \le \mathrm{a}$$

## Solution of Hyperbolic Equations

The domain of interest, $0 \le x \le a$ and $0 \le t \le b$, is partitioned as shown in Fig. 15.6. The rectangles of size $\Delta x = h$ and $\Delta t = \tau$

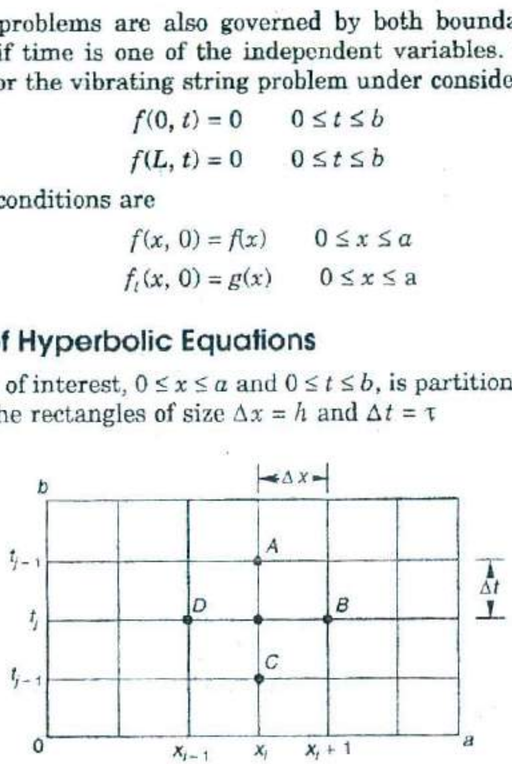


Fig. 15.6 Grid for solving hyperbolic equation

The difference equations for $f_{xx}(x, t)$ and $f_{tt}(x, t)$ are:

$$f_{xx}(x, t) = \frac{f(x-h, t) - 2f(x, t) + f(x+h, t)}{h^2}$$

$$f_{tt}(x, t) = \frac{f(x, t-\tau) - 2f(x, t) + f(x, t+\tau)}{\tau^2}$$

This implies that,

$$T\frac{f_{i-1,j}-2f_{i,j}+f_{i+1,j}}{h^2}=\rho\frac{f_{i,j-1}-2f_{i,j}+f_{i,j+1}}{\tau^2}$$

Solving this for $f_{i,j+1}$, we obtain

$$f_{i,j+1}=-f_{i,j-1}+2\left(1-\frac{T\tau^2}{\rho h^2}\right)f_{ij}+\frac{T\tau^2}{\rho h^2}(f_{i+1,j}+f_{i-1,j})$$

If we can make

$$1-\frac{T\tau^2}{\rho h^2}=0$$

then, we have

$$\boxed{f_{i,j+1}=-f_{i,j-1}+f_{i+1,j}+f_{i-1,j}} \tag{15.30}$$

The value of $f$ at $x = x_i$ and $t = t_j + \tau$ is equal to the sum of the values of $f$ at the point $x = x_i - h$ and $x = x_i + h$ at the time $t = t_j$ (previous time) minus the value of $f$ at $x = x_i$ at time $t = t_j - \tau$. From Fig. 15.6, we can say that,

$$f_A=f_B+f_D-f_C$$

## Starting Values

We need two rows of starting values, corresponding to $j = 1$ and $j = 2$ in order to compute the values at the third row. First row is obtained using the condition

$$f(x,0)=f(x)$$

The second row can be obtained using the second initial condition as follows:

$$f_t(x,0)=g(x)$$

We know that

$$f_t(x,0)=\frac{f_{i,0+1}-f_{i,0-1}}{2\tau}=g_i$$

$$f_{i,-1}=f_{i,1}-2\,\tau g_i \qquad \text{for } t=0 \text{ only}$$

Substituting this in Eq. (15.30), we get for $t = t_1$

$$\boxed{f_{i,1}=\frac{1}{2}(f_{i+1,0}+f_{i-1,0})+\tau g_i} \tag{15.31}$$

In many cases, $g(x_i) = 0$. Then, we have

$$f_{i,1}=\frac{1}{2}(f_{i+1,0}+f_{i-1,0})$$

### Example 15.7

Solve numerically the wave equation

$$f_{tt}(x, t) = 4f_{xx}(x, t), \qquad 0 \le x \le 5$$

with the boundary conditions

$$f(0,t) = 0 \quad \text{and} \quad f(5, t) = 0$$

and initial values

$$f(x, 0) = f(x) = x(5 - x)$$

$$f_t(x,0) = g(x) = 0$$

Let $h = 1$

Given,

$$\frac{T}{\rho} = 4$$

and assuming

$$1 - 4\frac{\tau^2}{1^2} = 0$$

We get,

$$\tau = \frac{1}{2}$$

The values estimated using Eqs (15.30) and (15.31) are tabulated in Table 15.4.

Table 15.4

| $t$ \ $x$ | *0* | *1* | *2* | *3* | *4* | 5 | |
|---|---|---|---|---|---|---|---|
| 0.0 | 0.0 | 4 | 6 | 6 | 4 | 0.0 | $x(5 - x)$ |
| 0.5 | 0.0 | 3 | 5 | 5 | 3 | 0.0 | Equation (15.31) |
| 1.0 | 0.0 | 1 | 2 | 2 | 1 | 0.0 | Equation (15.30) |
| 1.5 | 0.0 | −1 | −2 | −2 | −1 | 0.0 | |
| 2.0 | 0.0 | −3 | −5 | −5 | −3 | 0.0 | |
| 2.5 | 0.0 | −4 | −6 | −6 | −4 | 0.0 | |

## 15.6 SUMMARY

In this chapter, we discussed the solution of an important class of differential equations called partial differential equations. Due to complexity and limited scope of this book, we considered only the finite-difference method of solving the PDE problems where the coefficients $a$, $b$, and $c$ are constants. We presented the following in this chapter:

- Definition and classification of partial differential equations.
- Derivation of difference equations for PDEs.
- Solution of Laplace's equation by the method of elimination.

- Liebmann's iterative method for solving Laplace's equation.
- Solution of Poisson's equation by both direct and iterative methods.
- Solution of parabolic type heat equation using the explicit Bender-Schmidt recurrence equation and the implicit Crank-Nicholson formula.
- Solution of hyperbolic type wave equation by iterative procedure.

## Key Terms

*Bender-Schmidt equation*
*Crank-Nicholson formula*
*Cross average*
*Diagonal average*
*Dirichlet boundary condition*
*Elliptic equation*
*Explicit method*
*Finite-difference method*
*Finite-element method*
*Gauss-Seidel iteration*
*Heat equation*
*Hyperbolic equation*
*Implicit method*
*Laplace's equation*
*Laplacian operator*
*Liebmann's method*
*Parabolic equation*
*Partial derivatives*
*Partial differential equation*
*Poisson's equation*
*Wave equation*

## REVIEW QUESTIONS

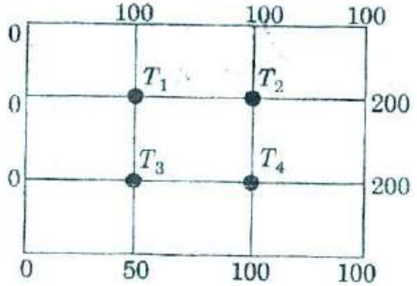
1. What is a partial differential equation? Give two examples.
2. State two real-life problems where partial differential equations are required to construct mathematical models.
3. How are the partial differential equations classified? Give an example from real-life situations for each type.
4. What are the various methods available to solve differential equations?
5. Explain how difference quotients are applied to solve partial differential equations.
6. What is Poisson's equation? How does it differ from Laplace's equation?
7. What is Liebmann's iteration method? What are its advantages?
8. What is meant by Dirichlet boundary conditions?
9. What is diagonal-averaging? When do we use it?
10. Derive a difference equation to represent a Poisson's equation.
11. Derive the five-point formula for Laplace's equation.
12. What is Crank-Nicholson method? Why is it known as implicit method?
13. What is Bender-Schmidt recurrence equation? Derive the formula.
14. Discuss the impact of size of the incremental width $\Delta T$ for the time variable $t$ on the solution of a heat-flow equation.
15. Outline the argument that demonstrates the stability of the finite-difference procedure for solving a hyperbolic equation.

## REVIEW EXERCISES

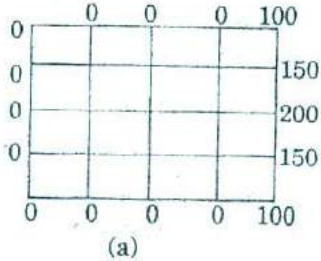
1. Determine which of the following equations are elliptic, parabolic, and hyperbolic.
   (a) $3f_{xx} + 4f_{yy} = 0$
   (b) $f_{xx} - f_{yy} = 0$
   (c) $f_{xx} - 2f_{xy} + 2f_{yy} = 2x + 5y$
   (d) $f_{xx} + 2f_{xy} + 4\,f_{yy} = 0$
   (e) $f_{xy} - f_{y} = 0$
   (f) $f_{xx} + 6f_{xy} + 9f_{yy} = 0$
2. The steady-state two-dimensional heat-flow in a metal plate is by

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0$$

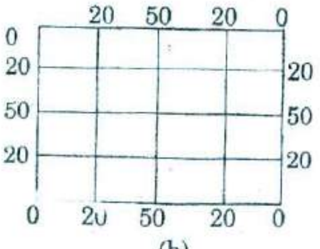
   Given the boundary conditions as shown in the figure below, find the temperatures $T_1$, $T_2$, $T_3$, and $T_4$.



3. Solve for the steady-state temperatures in a rectangular plate 8 cm × 10 cm, if one 10 cm side is held at 50°C, and the other 10 cm side is held at 30°C and the other two sides are held at 10°C. Assume square grids of size 2 cm × 2 cm.
4. Repeat Exercise 3 by Leibmann's method.
5. Evaluate $f(x, y)$ at the internal grid points of the given domain governed by Laplace's equation. Use Liebmann's iteration method.



(a)



(b)

6. Torsion on a rectangular bar subject to twisting is governed by

$$\nabla^2 T = -4$$

Given the condition $T = 0$ on boundary, find $T$ over a cross section of a bar of size 9 cm × 12 cm. Use a grid size of 3 cm × 3 cm.

7. Solve the equation

$$\nabla^2 f = F(x, y)$$

with $F(x, y) = xy$ and $f = 0$ on boundary. The domain is a square with corners at (0, 0) and (4, 4). Use $h = 1$.

8. Estimate the values at grid points of the following equations using Bender-Schmidt recurrence equation. Assume $h = 1$

   (a) $f_{xx} - 0.5 f_t = 0$

   Given,

   $f(0, t) = 0, f(5, t) = 0$

   $f(x, 0) = x(5 - x)$

   (b) $9f_{xx} = f_t$

   Given,

   $f(0, t) = -5, f(5, t) = 5$

$$f(x, 0) = \begin{cases} -5 \text{ for } 0 \le x \le 2.5 \\ 5 \text{ for } 2.5 < x \le 5 \end{cases}$$

9. Initial temperatures within an insulated cylindrical metal rod of 5 cm long are given by

$$T = 20x \text{ for } 0 \le x \le 5$$

where $x$ is the distance from one face. Both the ends are maintained at 0°C. Find the temperatures as a function of $x$ and $t$ if the heat flow is governed by

$$4T_{xx} - T_t = 0$$

10. Solve the following equation using Crank-Nicholson method.

$$\frac{\partial^2 f}{\partial x^2} = 8 \frac{\partial f}{\partial t}$$

Given,

$f(0, t) = 0, \qquad f(20, t) = 10$

$f(x, 0) = 2.0$

Assume $\Delta x = h = 5$ and $r = 1$

11. Solve Exercise 9 with the Crank-Nicholson method with $r = 1$.

12. Solve the following hyperbolic equations using finite difference method.

   (a) $f_{tt} = 4 f_{xx}$

   Given,

   $f(0, t) = 0$ and $f(5, t) = 0$

$f(x, 0) = 100x^2 (5 - x)$

$f_t (x, 0) = 0$

(b) $f_{tt} = 4 f_{xx}$

Given,

$f(0, t) = 0$ and $f(1, t) = 0$

$f(x, 0) = f(x) = \sin (\pi x) + \sin (2\pi x)$

$f_t (x,0) = 0$

## PROGRAMMING PROJECTS

1. Develop a program to solve Laplace's equation with Dirichlet conditions.
2. Write a program to solve Poisson's equation.
3. Develop a program using forward-difference method to solve the heat equation.
4. Write a program to solve the heat equation using Crank-Nicholson method.
5. Write a program for finite-difference solution of the wave equation.

APPENDIX A

# Solution of Linear Systems by Matrix Methods

## A.1 OVERVIEW OF MATRICES

A *matrix* is a rectangular array of *elements* arranged in rows and columns. If the matrix contains $m$ rows and $n$ columns, the matrix is said to be of size $m \times n$. The element in the $i$th row and $j$th column of the matrix $A$ is denoted by $a_{ij}$. For example,

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \\ a_{41} & a_{42} & a_{43} \end{bmatrix}$$

is a $4 \times 3$ matrix.

### Types of Matrices

Matrices may belong to one of the many types discussed here.

***1. Square Matrix*** A matrix in which rows $m$ is equal to columns $n$.

***2. Identity Matrix*** A square matrix in which all the diagonal elements are one and other elements are zero. That is

$$\begin{aligned} a_{ij} &= 1 \quad \text{for} \quad i = j \\ a_{ij} &= 0 \quad \text{for} \quad i \neq j \end{aligned}$$

Identity matrices are denoted by I. For example, a 3 × 3 identity is written as

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

***3. Row Vector*** A matrix with one row and $n$ columns.

***4. Column Vector*** A matrix with $m$ rows and one column.

***5. Transpose Matrix*** The matrix $A^T$ is called the transpose of $A$ if the element $a_{ij}$ in $A$ is equal to element $a_{ji}$ in $A^T$ for all $i$ and $j$. For example, if

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

then,

$$A^T = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

It is clear that $A^T$ is obtained by interchanging the rows and the columns of $A$.

***6. Zero Matrix*** A matrix in which every element is zero.

***7. Equal Matrix*** A matrix is said to be equal to another matrix if and only if they have the same order and the corresponding elements are equal. That is,

$$\text{matrix } A = \text{matrix } B$$

if $a_{ij} = b_{ij}$ for all $i$ and $j$.

## Matrix Algebra

We can perform only three operations, namely, addition, subtraction and multiplication on the matrices. The division, although not defined, is replaced by the concept of inversion discussed later.

Two matrices $A$ and $B$ can be added together (or subtracted from each other) if they are of the same order. Then

$$\text{sum } C = A + B$$

can be obtained by adding the corresponding elements. That is,

$$c_{ij} = a_{ij} + b_{ij} \qquad \text{for all } i \text{ and } j$$

Similarly,

$$\text{difference } E = A - B$$

can be obtained by subtracting the corresponding elements. That is,

$$e_{ij} = a_{ij} - b_{ij} \qquad \text{for all } i \text{ and } j$$

Two matrices $A$ and $B$ can be multiplied in the order $AB$ if and only if the number of columns of $A$ is equal to the number of rows of $B$. That is, if $A$ is of order $m \times r$, then $B$ should be of order $r \times n$, where $m$ and $n$ are arbitrary values. In such cases, we may obtain

$$P = AB$$

which is of order $m \times n$. Elements of the matrix $P$ is given by

$$p_{ij} = \sum_{k=1}^{r} a_{ik}\, b_{kj} \qquad \text{for all } i \text{ and } j$$

The following general properties apply to matrix algebra.

1. $A \pm B = B \pm A$
2. $A \pm (B \pm C) = (A \pm B) \pm C$
3. $(A \pm B)^T = A^T \pm B^T$
4. $IA = AI$, where $I$ is identity matrix
5. $(AB)C = A(BC)$
6. $C(A + B) = CA + CB$
7. $(A + B)C = AC + BC$
8. $\alpha(AB) = (\alpha A)B = A(\alpha B)$, where $\alpha$ is a scalar

## Traces and Determinants

The *trace* of a matrix is the number obtained by adding its diagonal elements (from upper left corner to the lower right corner). For example, the trace of the matrix

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \text{ is } a_{11} + a_{22} + a_{33}$$

The *determinant* of a $2 \times 2$ matrix, say $A$, is written in the form

$$\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix}$$

and the determinant is computed as

$$a_{11}\, a_{22} - a_{12}\, a_{21}$$

Similarly, for a $3 \times 3$ matrix the determinant is given by

$$|A| = \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix}$$

$$= a_{11} \begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} - a_{12} \begin{vmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{vmatrix} + a_{13} \begin{vmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{vmatrix}$$

$$= a_{11}\,(a_{22}a_{33} - a_{23}a_{32}) - a_{12}(a_{21}a_{33} - a_{31}a_{23})$$
$$+ a_{13}(a_{21}a_{32} - a_{22}a_{31})$$
$$= a_{11}a_{22}a_{33} + a_{12}$$ [illegible] $$a_{13}a_{21}a_{32} - a_{11}a_{32}a_{23}$$
$$- a_{21}a_{12}a_{33} - a_{31}a_{22}a_{13}$$

Note that there are 6 product terms added together. For larger matrices, the determinant is much more difficult to define and compute manually. In general, for an $n \times n$ matrix, the determinant will contain a sum of $n!$ signed product terms, each having $n$ elements.

Some of the important properties of determinants that would be helpful in computing their values are:

1. Interchanging two rows (or two columns) of a matrix changes the sign of the determinant but not the value.
2. If two rows (or columns) of a matrix are identical then its determinant is zero.
3. If a matrix contains a row (or column) of all zeros, then its determinant is zero.
4. Value of the determinant of a matrix does not change when a scalar multiple of one row (or column) is added to another row (or column).
5. If every element of a row (or column) is multiplied by a scalar $\alpha$, the value of the determinant is multiplied by $\alpha$.
6. If $A$ and $B$ are square matrices of same size, then $|AB| = |A|\,|B|$.
7. For a triangular matrix (in which all the elements below (or above) the diagonal are zero), the determinant is the product of the diagonal elements.

## Minors and Cofactors

The *minor* $M_{ij}$ of the element $a_{ij}$ of the determinant $|A|$ is obtained by striking out the $i$th row and $j$th column. That is,

$$M_{11} = \begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix}, \qquad M_{22} = \begin{vmatrix} a_{11} & a_{13} \\ a_{31} & a_{33} \end{vmatrix}$$

and so on. That is, the minor of a particular element is the determinant that remains after the row and the column that contain the element have been deleted.

The *cofactor* of an element is its minor with a sign attached. The cofactor $d_{ij}$ of an element $a_{ij}$ is given by

$$d_{ij} = (-1)^{i+j}\,M_{ij}$$

The value of the determinant of a matrix can be obtained by expanding the determinant by cofactors. This is done by choosing any column or row and determining the sum of the product of eac ı element in the chosen row or column and its cofactor.

### Adjoint Matrix

If $d_{ij}$ is the cofactor of the element $a_{ij}$ of the square matrix $A$, then, by definition, the *adjoint matrix* of $A$ is given by

$$\text{adj}(A) = D^T$$

where

$$D = \begin{bmatrix} d_{11} & d_{12} & \dots & d_{1n} \\ d_{21} & d_{22} & \dots & d_{2n} \\ \vdots & \vdots & & \\ d_{n1} & d_{n2} & \dots & d_{nn} \end{bmatrix}$$

### Inverse of a Matrix

If $B$ and $C$ are two $n \times n$ square matrices such that

$$BC = CB = I \text{ (Identity matrix)}$$

then, $B$ is called the *inverse* of $C$ and $C$ the inverse of $B$. The common notation for inverses is $B^{-1}$ and $C^{-1}$. That is,

$$B^{-1} B = I$$
$$C^{-1} C = I$$

The concept of matrix inversion is useful in solving linear systems of equations.

## A.2 SOLUTION OF LINEAR SYSTEMS BY DETERMINANTS

We can solve a system of linear equations by determinants using a method called *Cramer's Rule*. For the sake of simplicity, we consider a $2 \times 2$ system such as

$$ax_1 + bx_2 = c$$
$$dx_1 + ex_2 = f$$

We can solve for the variable $x_1$ by eliminating the variable $x_2$. Thus,

$$x_1 = \frac{ce - bf}{ae - bd}$$

Similarly,

$$x_2 = \frac{af - cd}{ae - bd}$$

Alternatively, we can express $x_1$ and $x_2$ using determinants. That is,

$$x_i = \frac{\begin{vmatrix} c & b \\ f & e \end{vmatrix}}{\begin{vmatrix} a & b \\ d & e \end{vmatrix}} \quad \text{and} \quad x_2 = \frac{\begin{vmatrix} a & c \\ d & f \end{vmatrix}}{\begin{vmatrix} a & b \\ d & e \end{vmatrix}}$$

This is known as *Cramer's rule*.

### Example A.1

Solve the following system of equations using Cramer's rule:

$$2x_1 + 3x_2 = 12$$

$$4x_1 - x_2 = 10$$

$$x_1 = \frac{\begin{vmatrix} 12 & 3 \\ 10 & -1 \end{vmatrix}}{\begin{vmatrix} 2 & 3 \\ 4 & -1 \end{vmatrix}} = \frac{-12-30}{-2-12} = \frac{42}{14} = 3$$

$$x_2 = \frac{\begin{vmatrix} 2 & 12 \\ 4 & 10 \end{vmatrix}}{\begin{vmatrix} 2 & 3 \\ 4 & -1 \end{vmatrix}} = \frac{20-48}{-14} = \frac{28}{14} = 2$$

## A.3 SOLUTION OF LINEAR SYSTEMS BY MATRIX INVERSION

A linear system of $n$ equations in $n$ unknowns can be represented in matrix form as

$$AX = B$$

where $A$, $X$ and $B$ are matrices and are given by

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

$$X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad B = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

$A$ is called *coefficient matrix* and $B$ is known as *constant vector*. $X$ is the required solution and, therefore, it is called the *solution vector*. If we multiply the matrix equation

$$AX = B$$

By $A^{-1}$ on both sides, we get

$$A^{-1}AX = A^{-1}B$$

$A^{-1}$ is the *inverse matrix* of $A$. We know that $A^{-1}A = I$ is the *identity matrix* and is given by

$$I = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & & & & \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix}$$

Therefore,

$$A^{-1}AX = IX = X$$

is the solution of the system of equations and is obtained from

$$X = A^{-1}B = CB$$

If we know the inverse of the matrix $A$, we can obtain the solution vector $X$ by post-multiplying it by $B$.

The inverse $A^{-1}$ of a square matrix $A$ exists, if and only if, $A$ is *nonsingular* (i.e. det $A \neq 0$). $A^{-1}$ is the matrix obtained from $A$ by replacing each element $a_{ij}$ by its *cofactor* $d_{ij}$ and then transposing the resulting matrix and dividing it by the *determinant* of $A$.

$$C = A^{-1} = \frac{\text{adj}(A)}{\det A}$$

where adj($A$) is the *adjoint* of matrix $A$ and is given by the transpose of the *cofactor* matrix of $A$.

Then

$$\text{adj}(A) = D^T$$

where

$$D = \begin{bmatrix} d_{11} & d_{12} & \dots & d_{1n} \\ d_{21} & d_{22} & \dots & d_{2n} \\ \vdots & \vdots & & \\ d_{n1} & d_{n2} & \dots & d_{nn} \end{bmatrix}$$

$d_{ij}$ is the cofactor of the element $a_{ij}$ and is given by

$$d_{ij} = (-1)^{i+j} M_{ij}$$

$M_{ij}$ is called the *minor* of $a_{ij}$ and is taken as the determinant of matrix $A$ after deleting $i$th row and $j$th column.

### Example A.2

**Solve the following system using matrix inversion method**

$$\begin{aligned} 2x_1 + x_2 + x_3 &= 7 \\ x_1 - x_2 + x_3 &= 0 \\ 4x_1 + 2x_2 - 3x_3 &= 4 \end{aligned}$$

Given, $$A = \begin{bmatrix} 2 & 1 & 1 \\ 1 & -1 & 1 \\ 4 & 2 & -3 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} 7 \\ 0 \\ 4 \end{bmatrix}$$

$$M_{11} = \begin{vmatrix} -1 & 1 \\ 2 & -3 \end{vmatrix} = 1 \qquad d_{11} = 1$$

$$M_{12} = \begin{vmatrix} 1 & 1 \\ 4 & -3 \end{vmatrix} = -7 \qquad d_{12} = 7$$

$$M_{13} = \begin{vmatrix} 1 & -1 \\ 4 & -2 \end{vmatrix} = 6 \qquad d_{13} = 6$$

$$M_{21} = \begin{vmatrix} 1 & 1 \\ 2 & -3 \end{vmatrix} = -5 \qquad d_{21} = 5$$

$$M_{22} = \begin{vmatrix} 2 & 1 \\ 4 & -3 \end{vmatrix} = -10 \qquad d_{22} = -10$$

$$M_{23} = \begin{vmatrix} 2 & 1 \\ 4 & 2 \end{vmatrix} = 0 \qquad d_{23} = 0$$

$$M_{31} = \begin{vmatrix} 1 & 1 \\ -1 & 1 \end{vmatrix} = 2 \qquad d_{31} = 2$$

$$M_{32} = \begin{vmatrix} 2 & 1 \\ 1 & 1 \end{vmatrix} = 1 \qquad d_{32} = -1$$

$$M_{33} = \begin{vmatrix} 2 & 1 \\ 1 & -1 \end{vmatrix} = -3 \qquad d_{33} = -3$$

$$D = \begin{bmatrix} 1 & 7 & 6 \\ 5 & -10 & 0 \\ 2 & -1 & -3 \end{bmatrix}$$

$$\text{Adj}A = D^T = \begin{bmatrix} 1 & 5 & 2 \\ 7 & -10 & -1 \\ 6 & 0 & -3 \end{bmatrix}$$

$$\det A = a_{11}d_{11} + a_{12}d_{12} + a_{13}d_{13} = 15$$

$$A^{-1} = C = \begin{bmatrix} \frac{1}{15} & \frac{5}{15} & \frac{2}{15} \\ \frac{7}{15} & \frac{-10}{15} & \frac{-1}{15} \\ \frac{6}{15} & 0 & \frac{-3}{15} \end{bmatrix}$$

$$B = \begin{bmatrix} 7 \\ 0 \\ 4 \end{bmatrix}$$

We know that $X = CB$ and therefore

$$x_1 = \frac{7}{15} + 0 + \frac{8}{15} = 1$$

$$x_2 = \frac{49}{15} - 0 - \frac{4}{15} = 3$$

$$x_3 = \frac{42}{15} + 0 - \frac{12}{15} = 2$$

## A.4 GAUSS-JORDAN MATRIX INVERSION

The Gauss-Jordan elimination technique (discussed in Chapter 7) can be used to invert a matrix effectively. The square matrix $A$ which is to be inverted is augmented by a unit matrix of the same order as follows.

$$\left[\begin{array}{cccc|cccc} a_{11} & a_{12} & \dots & a_{1n} & 1 & 0 & \dots & 0 \\ a_{21} & a_{22} & \dots & a_{2n} & 0 & 1 & \dots & 0 \\ \vdots & \vdots & & & \vdots & & & \\ a_{n1} & a_{n2} & \dots & a_{nn} & 0 & 0 & \dots & 1 \end{array}\right]$$

If we carry out Gauss-Jordan elimination using the first row as a pivot row, we get

$$\left[\begin{array}{cccc|cccc} 1 & a_{12} & \cdots & a_{1n} & a_{1,n+1} & 0 & \cdots & 0 \\ 0 & a_{21} & \cdots & a_{2n} & a_{2,n+1} & 1 & \cdots & 0 \\ \vdots & \vdots & & \vdots & \vdots & & & \\ 0 & a_{n2} & \cdots & a_{nn} & a_{n,n+1} & 0 & \cdots & 1 \end{array}\right]$$

When we repeat the process using the second row as the pivot row, the result is

$$\left[\begin{array}{ccccc|ccccc} 1 & 0 & a_{13} & \cdots & a_{1n} & a_{1,n+1} & a_{1,n+2} & 0 & \cdots & 0 \\ 0 & 1 & a_{23} & \cdots & a_{2n} & a_{2,n+1} & a_{2,n+2} & 0 & \cdots & 0 \\ 0 & 0 & a_{33} & \cdots & a_{3n} & a_{3,n+1} & a_{3,n+2} & 1 & \cdots & 0 \\ 0 & 0 & a_{n3} & \cdots & a_{nn} & a_{n,n+1} & a_{n,n+2} & 0 & \cdots & 1 \end{array}\right]$$

This elimination process, if continued for all the $n$ rows, yields the final result as follows:

$$\left[\begin{array}{ccccc|cccc} 1 & 0 & 0 & \cdots & 0 & a_{1,n+1} & a_{1,n+2} & \cdots & a_{1,n+n} \\ 0 & 1 & 0 & \cdots & 0 & a_{2,n+1} & a_{2,n+2} & \cdots & a_{2,n+n} \\ 0 & 0 & 1 & \cdots & 0 & \vdots & & & \\ \vdots & & & & & & & & \\ 0 & 0 & 0 & \cdots & 1 & a_{n,n+1} & a_{n,n+1} & \cdots & a_{n,n+n} \end{array}\right]$$

The matrix $a_{ij}$ for $i = 1$ to $n$ and $j = n + 1$ to $2n$ is the inverse matrix of $A$.

$$C = A^{-1} = [a_{ij}] \begin{cases} i = 1, \ldots, n \\ j = n + 1, \ldots, 2n \end{cases}$$

where the element

$$c_{ij} = a_{i,n+j} \text{ for } i = 1, \ldots, n \qquad \text{and} \qquad j = 1, \ldots, n.$$

### Example A.3

Find the inverse of the coefficient matrix of the system given in Example A2 using Gauss-Jordan elimination technique.

$$\text{Augmented } A = \left[\begin{array}{ccc|ccc} 2 & 1 & 1 & 1 & 0 & 0 \\ 1 & -1 & 1 & 0 & 1 & 0 \\ 4 & 2 & -3 & 0 & 0 & 1 \end{array}\right]$$

*Pivot row-1*

$$\left[\begin{array}{ccc|ccc} 1 & 1/2 & 1/2 & 1/2 & 0 & 0 \\ 0 & -3/2 & 1/2 & -1/2 & 1 & 0 \\ 0 & 0 & -5 & -2 & 0 & 1 \end{array}\right]$$

*Pivot row-2*

$$\left[\begin{array}{ccc|ccc} 1 & 0 & 2/3 & 1/3 & 1/3 & 0 \\ 0 & 1 & -1/3 & 1/3 & -2/3 & 0 \\ 0 & 0 & -5 & -2 & 0 & 1 \end{array}\right]$$

*Pivot row-3*

$$\left[\begin{array}{ccc|ccc} 1 & 0 & 0 & 1/15 & 1/3 & 2/15 \\ 0 & 1 & 0 & 7/15 & -2/3 & -1/15 \\ 0 & 0 & 1 & 2/5 & 0 & -1/5 \end{array}\right]$$

The last three columns represent the inverse of the matrix

$$\begin{array}{ccc} 2 & 1 & 1 \\ 1 & -1 & 1 \\ 4 & 2 & -3 \end{array}$$

Compare this with the result obtained in the previous section.

Once the inverse of $A$ is known, the solution vector $X$ can be obtained by simple matrix multiplication. That is

$$X = A^{-1}B = CB$$

$$x_1 = c_{11}b_1 + c_{12}b_2 + \ldots + c_{1n}b_n$$

$$x_2 = c_{21}b_1 + c_{22}b_2 + \ldots + c_{2n}b_n$$

and so on.

In general,

$$x_i = \sum_{j=1}^{n} c_{ij}b_j \qquad i = 1, \ldots, n$$

Note that $c_{ij} = a_{i,\,n+j}$ of augmented $A$ in the final step.

## Key Terms

*Adjoint matrix*
*Coefficient matrix*
*Cofactor*
*Column vector*
*Cramer's rule*
*Determinant*
*Equal matrix*
*Identity matrix*
*Inverse matrix*
*Minor*
*Nonsingular matrix*
*Row vector*
*Square matrix*
*Trace of a matrix*
*Transpose matrix*
*Zero matrix*

# APPENDIX B

# Solution of Polynomials by Graeffe's Root Squaring Method

Graeffe's root squaring method is a direct method of finding roots of a polynomial with real coefficients. This method deserves attention both for its historical interest and a novel idea involved.

Graeffe's method transforms a polynomial $p_n(x)$ into another polynomial of the same degree but whose roots are the squares of the roots of the original polynomial. Because of the squaring property, the roots of the new polynomial will be spread apart more widely than in the original one, when the roots are greater than 1 in absolute value. Repeating this process until the roots are really far apa , we can compute the roots directly from the coefficients.

We consider here a simple example to illustrate the root squaring technique.

Let

$$p_0(x) = (x - 1)(x - 2)(x - 3) \tag{B.1}$$

Then, we consider another function $p_1(y)$ such that

$$\begin{aligned} p_1(y) &= -p_0(x)\, p_0(-x) \\ &= -(x - 1)(x - 2)(x - 3)(-x - 1)(-x - 2)(-x - 3) \\ &= (x - 1)(x - 2)(x - 3)(x + 1)(x + 2)(x + 3) \\ &= (x^2 - 1)(x^2 - 4)(x^2 - 9) \\ &= (y - 1)(y - 4)(y - 9) \end{aligned} \tag{B.2}$$

where

$$y = x^2$$

We know that the roots of $p_0(x) = 0$ are $x_1 = 1$, $x_2 = 2$, and $x_3 = 3$. And from Eq. (B.2) the roots of $p_1(y) = 0$ are $y_1 = 1$, $y_2 = 4$, and $y_3 = 9$. It shows that roots of $p_1(y) = 0$ are the squares of the roots of $p_0(y) = 0$. This implies that if we compute the roots of $p_1(y)$, then we can obtain the roots of $p_0(x)$ from

$$x_1 = \sqrt{y_1}$$

$$x_2 = \sqrt{y_2}$$

$$x_3 = \sqrt{y_3}$$

Now, let us repeat the procedure for finding the roots of $p_1(y)$. Consider a third polynomial

$$\begin{aligned} p_2(z) &= -p_1(y)\, p_1(-y) \\ &= (z - 1)(z - 16)(z - 81) \qquad \text{(B.3)} \end{aligned}$$

The roots of Eq. (B.3) are

$$\begin{tikzcd} z_1 = 1\ (= y_1^2) & y_1\ (= x_1^2) \arrow[l] & x_1 \arrow[l] \\ z_2 = 16\ (= y_2^2) & y_2\ (= x_2^2) \arrow[l] & x_2 \arrow[l] \\ z_3 = 81\ (= y_3^2) & y_3\ (= x_3^2) \arrow[l, "\text{Iteration 2}"'] & x_3 \arrow[l, "\text{Iteration 1}"'] \end{tikzcd}$$

That is, after the second iteration, we can estimate the roots of original equation $p_0(x)$ from the relation

$$z_i = (x_i^2)^2 = x_i^4, \qquad i = 1, 2, 3$$

Suppose we have done the squaring process $k$ times and the roots of the final equation are $r_i$, then

$$\boxed{r_i = x_i^{(2^k)} = (x_i)^{2^k}} \qquad \text{(B.4)}$$

Remember that we never have $p_0(x)$ in factored form as given in Eq. (B.1), but the result is the same.

Now, let us consider a third degree polynomial in standard form as

$$p_0(x) = a_3x^3 + a_2x^2 + a_1x + a_0 \qquad \text{(B.5)}$$

Then,

$$\begin{aligned} p_1(y) &= -p_0(-x)\, p(x) \\ &= (a_3x^3 + a_2x^2 + a_1x + a_0)\,(a_3x^3 - a_2x^2 + a_1x - a_0) \\ &= a_3^2x^6 - (a_2^2 - 2a_1a_3)x^4 + (a_1^2 - 2a_0a_2)x^2 - a_0{}^2 \\ &= b_3y^3 + b_2y^2 + b_1y_1 + b_0 \qquad \text{(B.6)} \end{aligned}$$

where

$$y = x^2$$

and

$$\boxed{\begin{aligned} b_3 &= +\, a_3^2 \\ b_2 &= -(a_2^2 - 2a_1a_3) \\ b_1 &= +\, (a_1^2 - 2a_0a_2) \\ b_0 &= -{a_0}^2 \end{aligned}} \tag{B.7}$$

We can thus show that for a general polynomial of degree $n$,

$$p_0(x) = a_n x^n + a_{n-1}\, x^{n-1} + \dots + a_0 \tag{B.8}$$

after first squaring process,

$$\boxed{\begin{aligned} b_n &= +\, a_n^2 \\ b_{n-1} &= -(a_{n-1}^2 - 2a_na_{n-2}) \\ b_{n-2} &= +\, (a_{n-2}^2 - 2a_{n-1}\, a_{n-3} + 2a_na_{n-4}) \\ &\vdots \\ b_0 &= (-1)^n\, a_0^2 \end{aligned}} \tag{B.9}$$

This process can be repeated replacing '$a$' values by '$b$' values in Eq. (B.9) each time. Let us suppose that our final equation after $k$ iterations (i.e. squaring $k$ times) is

$$B_n y^n + B_{n-1} y^{n-1} + \dots + B_0 = 0 \tag{B.10}$$

Assuming that the roots of Eq. (B.10) are now more widely separated, we have

$$|y_1| >> |y_2| >> |y_3| \dots >> y_n$$

Then,

$$y_1 \approx -\frac{B_{n-1}}{B_n}$$

$$y_2 \approx -\frac{B_{n-2}}{B_{n-1}}$$

$$\vdots$$

$$y_n \approx -\frac{B_0}{B_1}$$

That is,

$$y_i \approx -\frac{B_{n-i}}{B_{n-i+1}},\ i = 1, 2, \dots, n$$

$$= (x_i)^{2^k} \tag{B.11}$$

Thus,

$$x_i = 2^k\text{th root of } y_i = \sqrt[2^k]{\left|\frac{B_{n-i}}{B_{n-i+1}}\right|} \tag{B.12}$$

The main advantage of Graeffe's root squaring method over other methods is that it does not require any initial guessing about roots. The method is also capable of giving all the roots but the limitation is that the polynomial should have only real coefficients.

### Example B.1

Apply root squaring technique to estimate the roots of

$$x^3 - 3x^2 - 6x + 8 = 0$$

Table below shows the coefficients of successive polynomials (using Eq. (B.7)) as well as the roots estimated (using Eq. (B.12)).

| $k$ | *Coefficients* | | | | *Roots estimated* | | |
|---|---|---|---|---|---|---|---|
| | $a_3$ | $a_2$ | $a_1$ | $a_0$ | $x_1$ | $x_2$ | $x_3$ |
| 0 | 1 | −3 | −6 | 8 | | | |
| 1 | 1 | −21 | 84 | −64 | 4.5826 | 2.0 | 0.8729 |
| 2 | 1 | −273 | 4368 | −4096 | 4.0648 | 2.0 | 0.9841 |
| 3 | 1 | −65793 | 16843008 | −16777216 | 4.0020 | 2.0 | 0.9995 |

The exact values are 1, −2, and 4. Signs must be determined by substituting the estimates in the original polynomial. When we substitute an estimated root, if the value of the polynomial is zero, then the root is positive; otherwise negative.

APPENDIX C

# Difference Operators and Central Difference Interpolation Formulae

## C.1 INTRODUCTION

In Chapter 9 we have already discussed briefly the application of finite differences for interpolating the function values. Here we consider again the finite differences of functions and discuss in detail various operators used on them. We also discuss here some of the central difference formulae used for interpolation.

## C.2 FINITE DIFFERENCES

Suppose we have a function, $f(x)$, whose values are known (or tabulated) at a set of points $x_0, x_1, x_2, \ldots, x_n$. Let us denote the function values $f(x_0)$, $f(x_1), \ldots, f(x_n)$ by $f_0, f_1, \ldots, f_n$. The difference between any two consecutive function values is called the *finite difference*. The difference in function values

$$f(x_{i+1}) - f(x_i) = f_{i+1} - f_i \tag{C.1}$$

is known as the *first forward difference* at $x = x_i$. We may denote this first difference at $x = x_i$ as

$$\boxed{\Delta f_i = f_{i+1} - f_i} \tag{C.2}$$

where $\Delta$ is an operator called the *forward difference operator*.

By applying the proces: repeatedly we generate the second forward difference, third forward difference, and so on. Thus the *k*th forward difference of $f(x)$ at $x = x_i$ is given by

$$\boxed{\Delta^k f_i = \Delta(\Delta^{k-1} f_i) = \Delta^{k-1} f_{i+1} - \Delta^{k-1} f_i} \tag{C.3}$$

**Example C.1**

Find expression for $\Delta^2 f_i$ and $\Delta^3 f_i$.

$$\begin{aligned}
\Delta^2 f_i &= \Delta(\Delta f_i)\\
&= \Delta(f_{i+1} - f_i)\\
&= \Delta f_{i+1} - \Delta f_i\\
&= f_{i+2} - f_{i+1} - f_{i+1} + f_i\\
&= f_{i+2} - 2f_{i+1} + f_i\\
\Delta^3 f_i &= \Delta(\Delta^2 f_i)\\
&= \Delta(f_{i+2} - 2f_{i+1} + f_i)\\
&= f_{i+3} - 3f_{i+2} + 3f_{i+1} - f_i
\end{aligned}$$

The difference in fι nction values

$$f(x_i) - f(x_{i-1}) = f_i - f_{i-1} \tag{C.4}$$

is known as the *first backward difference* at $x = x_i$. This is denoted by

$$\boxed{\nabla f_i = f_i - f_{i-1}} \tag{C.5}$$

where $\nabla$ is an operator called the *backward difference operator*.

The *k*th backward difference of $f(x)$ at $x = x_i$ is defined by

$$\boxed{\nabla^k f_i = \nabla(\nabla^{k-1} f_i) = \nabla^{k-1} f_i - \nabla^{k-1} f_{i-1}} \tag{C.6}$$

Forward and backward differences introduce asymmetry. Sometimes we may need formulae which are symmetrical about the points of interest. Such formulae are based on central differences.

The difference in function values

$$f_{i+1/2} - f_{i-1/2} \tag{C.7}$$

is known as the *first central difference* of $f(x)$ at $x = x_i$. This is denoted by

$$\boxed{\delta f_i = f_{i+1/2} - f_{i-1/2}} \tag{C.8}$$

where $\delta$ is called the *central difference operator*.

The *k*th central difference at $x = x_i$ is defined as

$$\boxed{\delta^k f_i = \delta(\delta^{k-1} f_i) = \delta^{k-1} f_{i+1/2} - \delta^{k-1} f_{i-1/2}} \tag{C.9}$$

Note that if the function values are only known at $x_1, x_2, \ldots, x_n$, then $f(x_{i+1/2})$ is indeterminate and therefore $\delta f(x_i)$ is not computable. However, we can evaluate $\delta^2 f(x_i)$ as follows:

$$\delta^2 f_i = \delta(f_{i+1/2} - f_{i-1/2})$$

$$= f_{i+1} - f_i - f_i + f_{i-1}$$

$$= f_{i+1} - 2f_i + f_{i-1} \tag{C.10}$$

In general, we can compute $\delta^{2k} f_i$ for all positive integers $k$.

## C.3 DIFFERENCE OPERATORS

We have seen three difference operators, namely, forward operator $\Delta$, backward operator $\nabla$, and central operator $\delta$. We consider here some more operators that are often used in the manipulation of interpolation formulae.

### Shift Operator

We have an operator known as *shift* or *displacement* or *translation operator* denoted by $E$. The displacement or shift operator is defined as

$$\boxed{Ef_i = f_{i+1}} \tag{C.11}$$

$$E^k f_i = f_{i+k} \tag{C.12}$$

If the values of $x_i$ are equally spaced such that

$$x_{i+1} - x_i = h$$

then

$$x_{i+1} = x_i + h$$

Eq. (C.12) may be written as

$$E^k f(x_i) = f(x_i + kh) \tag{C.13}$$

### Inverse Operator

An operator opposite of $E$ is known as *inverse operator* and is defined as

$$E^{-1} f(x) = g(x) \tag{C.14}$$

Note that

$$Eg(x) = EE^{-1} f(x) = f(x)$$

$$Eg(x) = g(x + h)$$

Then

$$g(x + h) = f(x)$$

Therefore,

$$g(x) = f(x - h)$$

That is,

$$\boxed{E^{-1} f(x) = f(x - h)} \tag{C.15}$$

Similarly,

$$E^{-k} f(x) = f(x - kh) \tag{C.16}$$

### Averaging Operator

The *cveraging operator* $\mu$ is defined as

$$\delta f_i = \frac{1}{2}\,(f_{i\,+\,1/2} + f_{i\,-\,1/2}) \qquad \text{(C.17)}$$

That is,

$$\mu f(x_i) = \frac{1}{2}\left[f\left(x_i + \frac{h}{2}\right) + f\left(x_i - \frac{h}{2}\right)\right]$$

## C.4 RELATIONS BETWEEN THE OPERATORS

The difference operators are related to one another in a number of ways. We consider here a few of them.

### Relation between Δ and E

We know that

$$\begin{aligned} \Delta f(x) &= f(x + h) - (f(x) \\ &= Ef(x) - 1.f(x) \\ &= (E - 1)\,f(x) \end{aligned}$$

Therefore,

$$\boxed{\Delta = E - 1 \qquad \text{or} \qquad E = \Delta + 1} \qquad \text{(C.18)}$$

### Relation between ∇ and *E*

$$\begin{aligned} \nabla f(x) &= f(x) - f(x - h) \\ &= 1.f(x) - E^{-1}f(x) \\ &= (1 - E^{-1})\,f(x) \end{aligned}$$

Therefore

$$\boxed{\nabla = 1 - E^{-1}} \qquad \text{(C.19)}$$

or

$$E = (1 - \nabla)^{-1}$$

(since $(E^{-1})^{-1} = E$)

### Relation between Δ, ∇ and E

We have

$$\begin{aligned} E\nabla &= E(1 - E^{-1}) \\ &= E - 1 \\ &= \Delta \end{aligned}$$

Therefore

$$\boxed{E\nabla = \Delta} \qquad \text{(C.20)}$$

Similarly, we can show that

$$E^{-1}\Delta = \nabla$$

## Relation between *E*, $\delta$ and $\mu$

We have

$$\delta f(x) = f\left(x + \frac{h}{2}\right) - f\left(x - \frac{h}{2}\right)$$
$$= E^{1/2}f(x) - E^{-1/2}f(x)$$

Therefore,

$$\boxed{\delta = E^{1/2} - E^{-1/2} = E^{-1/2}\Delta = E^{1/2}\nabla} \qquad \text{(C.21)}$$

Similarly, we can show that

$$\boxed{\mu = \frac{1}{2}(E^{1/2} + E^{-1/2})} \qquad \text{(C.22)}$$

From Eqs (C.21) and (C.22),

$$\mu\,\delta = \frac{1}{2}\,(E - E^{-1})$$

We know

$$\Delta + \nabla = (E - 1)\,(1 - E^{-1}) = E - E^{-1}$$

Then,

$$\mu\,\delta = \frac{1}{2}\,(\Delta + \nabla)$$

### Example C.2

Prove that $\Delta - \nabla = \Delta\nabla$.

$$\Delta - \nabla = (E - 1) - (1 - E^{-1})$$
$$= E - 2 + E^{-1}$$
$$\Delta\nabla = (E - 1)\,(1 - E^{-1})$$
$$= E - 2 + E^{-1}$$

Therefore

$$\Delta - \nabla = \Delta\nabla$$

### Example C.3

Show that $\Delta^2 = E^2 - 2E + 1$

$$\Delta^2 = (E - 1)\,(E - 1)$$
$$= E^2 - 2E + 1$$

### Example C.4

**Prove $\delta^2 = \Delta - \nabla$**

$$\delta^2 = [E^{1/2} - E^{-1/2}]^2$$
$$= E + E^{-1} - 2$$
$$\Delta - \nabla = (E - 1) - (1 - E^{-1})$$
$$= E + E^{-1} - 2$$

Hence,

$$\delta^2 = \Delta - \nabla$$

From Example C.2, we know

$$\Delta - \nabla = \Delta\nabla$$

and, therefore,

$$\delta^2 = \Delta\nabla$$

From Example C.3, we can prove that

$$\Delta^2 = E^2 - 2E + 1$$
$$= E(E - 2 + E^{-1})$$
$$= E\,(\Delta - \nabla) = E\Delta\nabla$$

Similarly, the readers may attempt to prove the following relations.

1. $\mu^2 = 1 + \frac{\delta^2}{4}$ or $\delta^2 = 4(\mu^2 - 1)$
2. $\frac{\Delta}{\nabla} - \frac{\nabla}{\Delta} = \Delta + \nabla$
3. $\frac{\Delta}{\nabla} - \frac{\nabla}{\Delta} = 2\mu\delta$

## C.5 CENTRAL DIFFERENCE INTERPOLATION FORMULAE

We have already discussed in Chapter 9 the forward and backward difference formulae. They are useful when the value required is at the beginning or at the end of the table. However, if the point of interest is located at the middle of the table, then we may use the formulae based on central differences.

Let $x_0$ be the middle point and $f_0$ the corresponding function value. Consider the values of $x$ on either side $x_0$ such that

$$f_1 = f(x_0 + h), \qquad f_{-1} = f(x_0 - h)$$
$$f_2 = f(x_0 + 2h), \qquad f_{-2} = f(x_0 - 2h)$$

and so on.

We can now form a difference table with the values $f(x)$ on either side of $x_0$, as shown in Table C.1.

**Table C.1** Central difference table

| $x$ | $f$ *difference* | *First difference* | *Second difference* | *Third difference* | *Fourth* |
|---|---|---|---|---|---|
| $x_0 - 2h$ | $f_{-2}$ | | | | |
| | | $\Delta f_{-2} = \delta f_{-3/2}$ | | | |
| $x_0 - h$ | $f_{-1}$ | | $\Delta^2 f_{-2} = \delta^2 f_{-1}$ | | |
| | | $\Delta f_{-1} = \delta f_{-1/2}$ | | $\Delta^3 f_{-2} = \delta^3 f_{-1/2}$ | |
| $x_0$ | $f_0$ | | $\Delta^2 f_{-1} = \delta^2 f_0$ | | $\Delta^4 f_{-2} = \delta^4 f_0$ |
| | | $\Delta f_0 = \delta f_{1/2}$ | | $\Delta^3 f_{-1} = \delta^3 f_{1/2}$ | |
| $x_0 + h$ | $f_1$ | | $\Delta^2 f_0 = \delta^2 f_1$ | | |
| | | $\Delta f_1 = \delta f_{3/2}$ | | | |
| $x_0 + 2h$ | $f_2$ | | | | |

The entries in the table are related using the relation between $\Delta$ and $\delta$ operators. We know that

$$\delta = \Delta E^{-1/2}$$

and therefore we can show that

$$\delta^2 f_{-1} = (\Delta E^{\ 1/2})^2 f_{-1} = \Delta^2 E^{-1} f_{-1} = \Delta^2 f_{-2}$$

Similarly, we can prove for all the entries.

We present here the following central difference interpolation formulae which use differences close to the centre of the table.

1. Gauss forward formula
2. Gauss backward formula
3. Stirling formula
4. Bessel formula
5. Laplace-Everett formula

## Gauss Forward Interpolation Formula

We have used the Newton-Gregory forward interpolation formula in Chapter 9. This is given by

$$f(x) = f_0 + p\Delta f_0 + \frac{p(p-1)}{2!}\Delta^2 f_0 + \frac{p(p-1)(p-3)}{3!}\Delta^3 f_0 + \dots \quad \text{(C.23)}$$

where

$$x = x_0 + ph \qquad \text{or} \qquad p = \frac{x - x_0}{h}$$

We can show that

$$\Delta^2 f_0 = \Delta^2 E f_{-1} = \Delta^2 (1 + \Delta) f_{-1} = \Delta^2 f_{-1} + \Delta^3 f_{-1}$$

$$\Delta^3 f_0 = \Delta^3 E f_{-1} = \Delta^3 (1 + \Delta) f_{-1} = \Delta^3 f_{-1} + \Delta^4 f_{-1}$$
$$\Delta^4 f_0 = \Delta^4 f_{-1} + \Delta^5 f_{-1} \qquad \text{and so on.}$$

Substituting for $\Delta^2 f_0$, $\Delta^3 f_0$, ..., the above equivalents in Eq. (C.23) and after simplification, we obtain

$$\boxed{f(x) = f_0 + \binom{p}{1}\Delta f_0 + \binom{p}{2}\Delta^2 f_{-1} + \binom{p+1}{3}\Delta^3 f_{-1} + \binom{p+1}{4}\Delta^4 f_{-1} + \ldots} \tag{C.24}$$

where

$$\binom{m}{n} = \frac{m(m-1)(m-2)\ldots(m-n+1)}{n!}$$

Equation (C.24) is known as *Gauss forward formula*. Note that this formula involves 'odd' differences below the centre line and 'even' differences on the central line.

## Gauss Backward Interpolation Formula

Gauss backward formula is obtained using the odd differences above the centre line and even differences on the central line. This is obtained by using the following substitutions in Eq. (C.23).

$$\Delta f_0 = \Delta f_{-1} + \Delta^2 f_{-1}$$
$$\Delta^2 f_0 = \Delta^2 f_{-1} + \Delta^3 f_{-1}$$
$$\Delta^3 f_0 = \Delta^3 f_{-1} + \Delta^4 f_{-1}$$
$$\Delta^3 f_{-1} = \Delta^3 f_{-2} + \Delta^4 f_{-2}$$
$$\Delta^4 f_{-1} = \Delta^4 f_{-2} + \Delta^5 f_{-2}$$

Thus, we obtain the Gauss backward interpolation formula as

$$\boxed{f(x) = f_0 + \binom{p}{1}\Delta f_{-1} + \binom{p+1}{2}\Delta^2 f_{-1} + \binom{p+1}{3}\Delta^3 f_{-2} + \binom{p+2}{4}\Delta^4 f_{-2} + \ldots} \tag{C.25}$$

Equation (C.25) is popularly known as *Gauss backward formula.*

## Stirling Formula

Stirling formula is obtained by taking the average of the two Gauss formulae. Therefore, adding Eqs (C.24) and (C.25) and dividing by 2, we obtain

$$\boxed{\begin{aligned} f(x) = f_0 &+ p\left(\frac{\Delta f_0 + \Delta f_{-1}}{2}\right) + \frac{p^2}{2!}\Delta^2 f_{-1} \\ &+ \frac{p(p^2-1)}{3!}\left(\frac{\Delta^3 f_{-1} + \Delta^3 f_{-2}}{2}\right) + \frac{p^2(p^2-1)}{4!}\Delta^4 f_{-2} + \ldots \end{aligned}} \tag{C.26}$$

Equation (C.26) is known as *Stirling formula*. Note that Eq. (C.26) involves the means of the 'odd' differences just above and just below the central line and 'even' differences on the line. To use this formula, $p$ must satisfy the condition

$$-\frac{1}{2} < p < \frac{1}{2}$$

## Bessel Formula

Bessel formula is a modified form of Gauss forward formula, which is given below.

$$f(x) = f_0 + p\Delta f_0 + \frac{p(p-1)}{2!}\Delta^2 f_{-1} + \frac{(p+1)\,p(p-1)}{3!}\Delta^3 f_{-1} + \dots \quad \text{(C.27)}$$

We know

$$\Delta f_0 = f_1 - f_0 \quad \text{and, therefore,} \quad f_0 = f_1 - \Delta f_0 \quad \text{(C.28)}$$

$$\Delta^3 f_{-1} = \Delta^2 f_0 - \Delta^2 f_{-1}, \quad \text{then,} \quad \Delta^2 f_{-1} = \Delta^2 f_0 - \Delta^3 f_{-1} \quad \text{(C.29)}$$

Similarly,

$$\Delta^4 f_{-2} = \Delta^4 f_{-1} - \Delta^5 f_{-2} \quad \text{(C.30)}$$

Now, we rewrite Eq. (C.27) as

$$f(x) = \left(\frac{f_0}{2} + \frac{f_0}{2}\right) + p\Delta f_0 + \frac{1}{2}\frac{p(p-1)}{2!}\Delta^2 f_{-1} + \frac{1}{2}\frac{p(p-1)}{2!}\Delta^2 f_{-1}$$

$$+ \frac{(p+1)\,p(p-1)}{3!}\Delta^3 f_{-1} + \dots \quad \text{(C.31)}$$

Now, substituting Eqs (C.28), (C.29), and (C.30) in (C.31), we get

$$f(x) = \frac{f_0}{2} + \frac{f_1 - \Delta f_0}{2} + p\Delta f_0 + \frac{1}{2}\frac{p(p-1)}{2!}\Delta^2 f_{-1}$$

$$+ \frac{1}{2}\frac{p(p-1)}{2!}(\Delta^2 f_0 - \Delta^3 f_{-1}) + \frac{(p+1)\,p(p-1)}{3!}\Delta^3 f_{-1} + \dots$$

$$= \frac{f_0 + f_1}{2} + \left(p - \frac{1}{2}\right)\Delta f_0 + \frac{p(p-1)}{2!}\frac{(\Delta^2 f_0 + \Delta^2 f_{-1})}{2}$$

$$+ \frac{\left(p - \frac{1}{2}\right)p(p-1)}{3!}(\Delta^3 f_{-1}) + \dots \quad \text{(C.32)}$$

Equation (C.32) is called *Bessel formula*. Note that the Bessel formula involves odd differences below the centre line and averages of the even differences on and below the central line.

Observe that when $p = \frac{1}{2}$ all odd order differences vanish and we get

$$f(x) = \frac{1}{2}(f_0 + f_1) - \frac{1}{8}\left(\frac{\Delta^2 f_{-1} + \Delta^2 f_0}{2}\right) + \frac{3}{128}\left(\frac{\Delta^4 f_{-2} + \Delta^4 f_{-1}}{2}\right)$$

$$-\frac{5}{1024}\left(\frac{\Delta^6 f_{-3} + \Delta^6 f_{-2}}{2}\right) + \ldots \qquad \text{(C.33)}$$

Equation (C.33) is known as *formula for interpolating to haves.*

## Laplace-Everett Formula

Again consider the Gauss forward formula

$$f(x) = f_0 + p\Delta f_0 + \frac{p(p-1)}{2!}\Delta^2 f_{-1} + \frac{(p+1)\,p(p-1)}{3!}\Delta^3 f_{-1} + \ldots \qquad \text{(C.34)}$$

If we eliminate the add differences from Eq. (C.34) the result will give us the Laplace-Everett formula. We know

$$\Delta f_0 = f_1 - f_0$$
$$\Delta^3 f_{-1} = \Delta^2 f_0 - \Delta^2 f_{-1}$$
$$\Delta^5 f_{-2} = \Delta^4 f_{-1} - \Delta^4 f_{-2}$$

and so on.

Substituting these in Eq. (C.34) and rearranging the terms, we get

$$f(x) = (1-p)f_0 + \binom{p}{1} f_1 - \binom{p}{1}\Delta^2 f_{-1} + \binom{p+1}{3}\Delta^2 f_0$$

$$-\binom{p+1}{5}\Delta^4 f_{-2} + \binom{p+2}{5}\Delta^4 f_{-1} + \ldots \qquad \text{(C.35)}$$

Letting

$$p = 1 - q \qquad \text{or} \qquad 1 - p = q$$

and simplifying, Eq. (C.35) becomes

$$f(x) = \left[qf_0 + \binom{q+1}{3}\Delta^2 f_{-1} + \ldots\right] + \left[pf_1 + \binom{p+1}{3}\Delta^2 f_0 + \ldots\right] \qquad \text{(C.36)}$$

Equation (C.36) is known as Laplace-Everett formula. This formula involves only even differences on and below the central line. This can be used when $0 < p < 1$.

## Key Terms

*Averaging operator*
*Backward difference*
*Backward difference operator*
*Bessel formula*
*Central difference*
*Central difference operator*
*Displacement operator*
*Forward difference*
*Forward difference operator*
*Gauss backward formula*
*Gauss forward formula*
*Inverse operator*
*Laplace-Everett formula*
*Shift operator*
*Stirling formula*
*Translation operator*

APPENDIX D

# C Programs

## D.1 INTRODUCTION

C programs are basically made of functions. As we know, a function is a unit of a program that performs a particular task. A C function is similar to a subroutine in FORTRAN. A function begins running when its name is used in the program.

All C programs need a *main* function which is named as **main()**. Therefore, the programs in this appendix begin with the main function. Since the main function does not use any arguments, its name is followed by a set of empty parentheses.

Everything inside a C function is enclosed between two curly brackets, like {...}. The left curly bracket signals the beginning of the function and the right one signals the end of the function.

Some features of C that are distinctly, different from FORTRAN are:

- C is a free-form language and therefore we may follow any format of statements that may please us.
- A comment or remark can be inserted anywhere that a space can appear in a C program provided it is preceded by /* and followed by */
- As a convention (and not as a rule), everything in C is written using lower-case letters. Thus, names of all variables and functions are written in lower-case.
- All C statements must end with a semicolon.
- C does not support statement numbers. However, we can assign a label to a statement as follows:

```
begin : sum = 0.0;
```

We may direct the control to this statement by using the **goto** statement as

```
goto begin;
```

- C language does not have STOP and END statements. While the closing bracket } represents the "END" of a function, STOP may be replaced by invoking the **exit()** function available in C.
- Unlike FORTRAN, all variables and functions that return values must explicitly be declared for their types.

```
int i,j,a,b; /* declaring integer variables */
float x,y,a[10]; /*declaring real variables */
```

- C supports what are known as preprocessor directives **#include** and **#define**. The **#include** directive is used to include in the program library functions and other files. We may use the **#define** directive to define constants and functions that are used in the program.
- C does not support any operator for exponentiation. Instead, it uses a function **pow(x,y)** to compute $x^y$.
- C uses square brackets to represent arrays variables, like, $a[i]$. $b[i][j]$, etc.
- Unlike FORTRAN, C array elements are numbered from ZERO, not ONE. That is, the array $a[3]$ will be represented in the memory as $a[0]$, $a[1]$, and $a[2]$.
- C uses an operator & known as address operator to read the values from the keyboard. We may use this operator to obtain the address of a variable in the memory. For example, & $x$ gives the address of the location of $x$.
- C defines % as the modules operator. This operator divides the first operand by the second and gives the remainder, (not the quotient) as the result.
- C defines another operator known as "star" operator * to define a pointer. The statements

```
int a;
int *p; /* p is declared as pointer */
p=&a;    /* address of a is assigned to p */
```

make **p** point to **a**. Now, the statements *p+5 and a+5 both would give the same result. That is, ***p** means the value stored at the address pointed to by **p**.

- C uses the following relational and logical operators:

= = is equal to
< is less than
> is greater than
<= is less than or equal to
>= is greater than or equal to
!= is not equal to
|| logical OR
&& logical AND

- C has two more control statements **continue** and **break in addition** to the conventional **goto** statement. These two are **used inside a** loop. While the break transfers the control to the first **statement** outside the loop, the continue statement skips the remaining **part** of the loop.

## D.2 FORTRAN TO C CONVERSION

The FORTRAN programs in the text have been translated almost line by line using C equivalents given in Table D.1

Table D.1 C equivalents

| *FORTRAN Statements* | *C Equivalents* |
|---|---|
| INTEGER | int |
| REAL | float, double |
| PARAMETERS (M = 100) | #define M 100 |
| INTRINSIC | #include <math.h> |
| READ(*,*) statement | scanf() statement |
| WRITE (*,*) statement | printf() statement |
| Function subprograms | Preprocessor macro definition |
| Statement number | Label name |
| GOTO <number> | goto <label> |
| GOTO inside a loop | break or continue |
| STOP inside a program | exit() function |
| IF (...) THEN... | if (...) ...; |
| IF(...) THEN ... ELSE | if (...) ... else |
| DO ... CONTINUE | for (...) {...} |
| END | } |

The preprocessor directives **#define** and **#include** are always placed before the main() function. Any constants and variables declared before the main function are considered to be global and are available to all the functions defined after that point in the file.

It should be noted that the programs have been translated line by line so as to enable the readers to understand the logic easily. It is quite possible to simplify or improve the efficiency of C programs in this appendix using some of the unique features of C. This requires a complete understanding of the language.

Readers new to C language must read an introductory book on C such as *Programming in ANSI C* by E Balagurusamy before attempting to analyse and apply the C programs given in this appendix.

Table D.2 List of programs

| *Program No.* | *Name* | *Description* |
|---|---|---|
| 1 | POLY | Program POLY evaluates a polynomial of degree n at any point x using Horner's rule. |
| 2 | BISECT | This program finds a root of a nonlinear equation using the bisection method. |
| 3 | FALSE | This program finds a root of a nonlinear equation by false position method. |
| 4 | NEWTON | This program finds a root of a nonlinear equation by Newton-Raphson method. |
| 5 | SECANT | This program finds a root of a nonlinear equation by secant method. |
| 6 | FIXEDP | This program finds a root of a function using the fixedp point iteration method. |
| 7 | MULTIR | The program finds all the real roots of a polynomial. |
| 8 | COMPR | This program locates all the roots, both real and complex, using the Bairstow method. |
| 9 | MULLER# | This program evaluates root of a polynomial using Muller's method. |
| 10 | LEG1 | This program solves a system of linear equations using simple Gaussian elimination method. |
| 11 | LEG2 | This program solves a system of linear equations using Gaussian elimination with partial pivoting. |
| 12 | DOLIT | This program solves a system of linear equations using Dolittle LU decomposition. |
| 13 | JACIT | This program uses the subprogram JACOBI to solve a system of equations by Jacobi iteration method. |
| 14 | GASIT | This program uses the subprogram GASEID to solve a system of equations by Gauss-Seidel iteration method. |
| 15 | LAGRAN | This program computes the interpolation value at a specified point, given a set of data points, using the Lagrange interpolation representation. |
| 16 | NEWINT | This program constructs the Newton interpolation polynomial for a given set of data points and then computes interpolation value at a specified value. |
| 17 | SPLINE | This program computes the interpolation value at a specified value, given a set of table points, using the natural cubic spline interpolation. |
| 18 | LINREG | This program fits a line Y = A + BX to a given set of data points by the method of least squares. |
| 19 | POLREG | This program fits a polynomial curve to a given set of data points by the method of least squares. |
| 20 | NUDIF# | This program computes the derivative of a tabulated function at a specified value using the Newton interpolation approach. |

*(Contd.)*

**Table D.2** *(Contd.)*

| *Program No.* | *Name* | *Description* |
|---|---|---|
| 21 | TRAPE1 | This program integrates a given function using the trapezoidal rule. |
| 22 | SIMS1 | This program integrates a given function using the Simpson's 1/3 rule. |
| 23 | ROMBRG | This program performs Romberg integration by bisecting the intervals N times. |
| 24 | TRAPE2[#] | This program integrates a tabulated function using the trapezoidal rule. |
| 25 | SIMS2[#] | This program integrates a tabulated function using the Simpson's 1/3 rule. If the number of segments is odd, the trapezoidal rule is used for the last segment. |
| 26 | EULER | This program estimates the solution of the first order differential equation $y' = f(x, y)$ at the given point using Euler's method. |
| 27 | HEUN | This program solves the first order differential equation $y' = f(x, y)$ using the Heun's method. |
| 28 | POLYGN | This program solves the differential equation of type $y' = f(x, y)$ by polygon method. |
| 29 | RUNGE4 | This program computes the solution of first order differential equation of type $y' = f(x, y)$ using the 4th order Runge-Kutta method. |
| 30 | MILSIM | This program solves the first order differential equation $y' = f(x, y)$ using the Milne-Simpson method. |

[#] These are new programs in C and are not available in FORTRAN version in the text. Readers may try to develop FORTRAN equivalents of these programs.

## Program 1 POLY

```
/*-----------------------------------------------------------*
 *                                                           *
 * Main program                                              *
 *    Program POLY evaluates a polynomial of degree n        *
 *    at any point X using Horner's rule                     *
 *-----------------------------------------------------------*
 *                                                           *
 * Subroutines used                                          *
 *    horner                                                 *
 *-----------------------------------------------------------*
 *                                                           *
 * Variables used                                            *
 *    n - Degree of polynomial                               *
 *    a - Array of polynomial coefficients                   *
 *    x - Point of evaluation                                *
 *    p - Value of polynomial at x                           *
 *-----------------------------------------------------------*
 *                                                           *
 * Constants used                                            *
 *    NIL                                                    *
 *-----------------------------------------------------------*/
```

```
main( )
{
    int n,i;      /* Declaration of variables */
    float x,p,a[10];
    float horner(int n, float a[], float x);
    printf("Input degree of polynomial, n\n");
    scanf("%d", &n);
    printf("Input polynomial coefficients a(0) to a(n) \n");
    for(i=0;i<=n;i++)
      scanf("%f", &a[i]);
    printf("Input value of x (point of evaluation) \n");
    scanf("%f", &x);
/* Evaluating polynomial at x using Horner's rule */
    p = horner(n,a,x); /* Calling the function horner */
/* Writing the result */
      printf("\n");
    printf("f(x) = %f at x = %f \n", p, x);
    printf("\n");
}
/* End of main program POLY */
/* ---------------------------------------------------------- */
/* Defining the function horner */
float horner(int n, float a[], float x)
/*   horner computes the value of a polynomial of order n
     at any given point x. */
    {
        int i; /* Local variables */
        float p;
        p = a[n];
        for(i=n-1;i>=0;i–)
        {
          p = p*x + a[i];
        }
        return(p);
      }
/* End of Function horner */
/* ---------------------------------------------------------- */
```

## Program 2 BISECT

```
/* ------------------------------------------------------ *
 *  Main program                                          *
 *    This program finds a root of a nonlinear            *
 *    equation using the bisection method                 *
 * ------------------------------------------------------ *
 *  Functions invoked                                     *
 *    Macro F(x)                                          *
 * ------------------------------------------------------ *
 *  Subroutines used                                      *
 *    bim()                                               *
 * ------------------------------------------------------ *
 *  Variables used                                        *
 *    a - Left endpoint of interval                       *
 *    b - Right endpoint of interval                      *
 *    s - Status                                          *
 *    root - Final Solution                               *
 *    count - Number of Iterations done                   *
 * ------------------------------------------------------ *
 *  Constants used                                        *
 *    EPS - Error bound                                   *
 * ------------------------------------------------------ */
#include <stdio.h>
#include          <math.h>
#define EPS       0.000001
#define F(x)      (x)*(x)+(x)-2

main( )
{
    int s, count;
    float a,b,root;

    printf("\n");
    printf("SOLUTION BY BISECTION METHOD \n");
    printf("\n");
    printf("Input starting values \n");
    scanf("%f %f",&a,&b);

/* Calling the subroutine bim() */

    bim(&a, &b, &root, &s, &count);

    if(s==0)
    {
      printf("\n");
      printf("Starting points do not bracket any root \n");
      printf("(Check whether they bracket EVEN roots \n");
      printf("\n");
    }
    else
    {
      printf("\nRoot = %f \n", root);
      printf("F(root) = %f \n", F(root));
      printf("\n");
      printf("Iterations = %d \n", count);
      printf("\n");
    }
}
/* End of main program */

/* ----------------------------------------------------- */

/* Defining the subroutine bim() */

bim(float *a, float *b, float *root, int *s, int *count)

/* This subroutine finds a root of nonlinear equation
   in the interval [a,b] using the bisection method */
{
      float x1,x2,x0,f0,f1,f2;
/* Function values at initial points */
      x1 = *a;
      x2 = *b;
      f1 = F(x1);
      f2 = F(x2);
/* Test if initial values bracket a SINGLE root */
    if(f1*f2 > 0)
    {
      *s = 0;
      return;                   /* Program terminated */
    }
    else
/* Bisect the interval and locate the root iteratively */
    {
      *count = 0;

      begin:                    /* Iteration begins */
      x0 = (x1 + x2)/2.0;
      f0 = F(x0);
      if(f0==0)
      {
        *s = 1;
        *root = x0;
        return;
      }
      if(f1*f0 < 0)
      {
          x2 = x0;
      }
      else
      {
          x1 = x0;
          f1 = f0;
      }
/* Test for accuracy and repeat the process, if necessary */
      if(fabs((x2-x1)/x2) < EPS)
      {
          *s = 1;
          *root = (x1+x2)/2.0;
          return;                 /* Iteration ends */
      }
      else
      {
          *count = *count + 1;
          goto begin;
      }
     }
}
/* End of subroutine bim() */
/* ---------------------------------------------------- */
```

## Program 3 FALSE

```
/* ------------------------------------------------------ *
 *  Main program                                          *
 *    This program finds a root of a nonlinear equation   *
 *    by false position method                            *
 * ------------------------------------------------------ *
 *  Functions invoked                                     *
 *    Macro F(x)                                          *
 * ------------------------------------------------------ *
 *  Subroutines used                                      *
 *    fal()                                               *
 * ------------------------------------------------------ *
 *  Variables used                                        *
 *    a - Left endpoint of interval                       *
 *    b - Right endpoint of interval                      *
 *    s - Status                                          *
 *    root - Final solution                               *
 *    count - Number of iterations completed              *
 * ------------------------------------------------------ *
 *  Constants used                                        *
 *    EPS - Error bound                                   *
 * ------------------------------------------------------ */
```

```
#include  <math.h>
#define EPS      0.000001
#define F(x)     (x)*(x)+(x)-2
main( )
{
      int s,count;
      float a,b,root;
      printf("\n");
      printf("SOLUTION BY FALSE POSITION METHOD \n");
      printf("\n");
      printf("Input starting values \n");
      scanf("%f %f", &a, &b);
/* Calling the function fal() */
      fal(&a,&b,&s,&root,&count);
      if(s==0)
      {
          printf("\n");
          printf("Starting points do not bracket
            any root \n");
          printf("\n");
      }
      else
      {
          printf("\n");
          printf("Root = %f \n", root);
          printf("F(root) = %f \n", F(root));
          printf("\n NO. OF ITERATIONS = %d \n", count);
      }
}
/* End of main() program */
/* ----------------------------------------------------- */
/* Defining the subroutine fal() */
fal(float *a, float *b, int *s, float *root, int *count)
/* fal finds a root of a nonlinear equation */
{
      float x1,x2,x0,f0,f1,f2;
      x1 = *a;
      x2 = *b;
      f1 = F(x1);
      f2 = F(x2);
/* Test if a and b bracket a root */
```

```
      if(f1*f2 > 0)
      {
          *s = 0;
          return;                 /* Program terminated */
      }
      else
      {
          printf("       n          x1            x2 \n");
      }
      *count = 1;
      begin:                       /* Iteration begins */
      x0 = x1-f1*(x2-x1)/(f2-f1);
      f0 = F(x0);
      if(f1*f2 < 0)
      {
          x2 = x0;
          f2 = f0;
      }
      else
      {
          x1 = x0;
          f1 = f0;
      }
      printf("%5d %15.6f %15.6f \n", *count, x1, x2);
/* Test whether the desired accuracy has been achieved */
      if(fabs((x2-x1)/x2) < EPS)
      {
          *s = 1;
          *root = (x1+x2)*0.5;
          return;                    /* Iteration ends */
      }
      else
      {
          *count = *count + 1;
          goto begin;
      }
}
/* End of subroutine fal() */
/* ---------------------------------------------- */
```

## Program 4 NEWTON

```
/* -----------------------------------------------  *
 * Main program                                     *
 *   This program finds a root of a nonlinear equation  *
 *   by Newton-Raphson method                       *
 * -------------------------------------------------------- *
 * Functions invoked                                         *
 *   Macros F(x), FD(x), Library function fabs()             *
 * -------------------------------------------------------- *
 * Subroutines used                                          *
 *   NIL                                                     *
 * -------------------------------------------------------- *
 * Variables used                                            *
 *   x0 - Initial vale of x                                  *
 *   xn - New value of x                                     *
 *   fx - Function value at x                                *
 *   fdx - Value of function derivative at x                 *
 *   count - Number of iterations done                       *
 * -------------------------------------------------------- *
 *   Constants used                                          *
 *     EPS - Error bound                                     *
 *     MAXIT - Maximum number of iterations permitted        *
 * -------------------------------------------------------- */

#include <math.h>
#define EPS 0.000001
#define MAXIT 20
#define F(x)  (x)*(x)+(x)-2
#define FD(x)  2*(x)+1

main( )
{
     int count;
     float x0, xn, fx, fdx;

     printf("\n");
     printf("Input initial value of x \n");
     scanf("%f", &x0);
     printf("\n");
     printf("     SOLUTION BY NEWTON-RAPHSON METHOD \n");
     printf("\n");

     count = 1;
     begin:                          /* Iteration begins */
     fx = F(x0);
     fdx = FD(x0);
     xn = x0 - fx/fdx;

     if(fabs((xn-x0)/xn) < EPS)      /* Iteration ends */
     {
        printf("Root = %f \n", xn);
        printf("Function value = %f \n", F(xn));
        printf("Number of iterations = %d \n", count);
        printf("\n");
```

```
   }
   else
 {
    x0 = xn;
    count = count + 1;
    if(count < MAXIT)
    {
        goto begin;
    }
    else
    {
        printf("\n SOLUTION DOES NOT CONVERGE \n");
        printf("IN %d ITERATIONS \n", MAXIT);
    }
  }
}
/* End of main() program */
/* ------------------------------------------------------- */
```

## Program 5 SECANT

```
/* ------------------------------------------------------- *
* Main program                                              *
*   This program finds a root of a nonlinear                *
*   equation by secant method                               *
* ------------------------------------------------------- *
* Functions invoked                                         *
*   Macro F(x)                                              *
* ------------------------------------------------------- *
* Subroutines used                                          *
*   sec()                                                   *
* ------------------------------------------------------- *
*Variables used                                             *
*     a - Left endpoint of interval                         *
*     b - Right endpoint of interval                        *
*    x1 - New left point                                    *
*    x2 - New right point                                   *
*  root - Final solution                                    *
* count - Number of iterations completed                    *
* ------------------------------------------------------- *
* Constants used                                            *
*   EPS - Error bound                                       *
*   MAXIT - Maximum number of iterations permitted          *
* ------------------------------------------------------- */

#include  <math.h>
#define   EPS     0.000001
```

```
#define   MAXIT   50
#define   F(x)    (x)*(x)+(x)-2
main( )
{
     float a,b,root,x1,x2;
     int count,status;

     printf("\n");
     printf("        SOLUTION BY SECANT METHOD \n");
     printf("\n");
     printf("Input two starting points \n");
     scanf("%f %f", &a, &b);

     sec(&a, &b, &x1, &x2, &root, &count, &status);

     if(status == 1)
     {
        printf("\n");
        printf("     DIVISION BY ZERO \n");
        printf("\nLast x1 = %f \n", x1);
        printf("\nLast x2 = %f \n", x2);
        printf("\nNO. OF ITERATIONS = %d \n", count);
        printf("\n");
     }
     else if(status == 2)
     {
        printf("\n");
        printf("NO CONVERGENCE IN %d ITERATIONS \n", MAXIT);
        printf("\n");
     }
     else
     {
        printf("\n");
        printf("Root = %f \n", root);
        printf("Function value at root = %f \n", F(root));
        printf("\n");
        printf("NO. OF ITERATIONS = %d \n", count);
        printf("\n");
     }
}
/* End of main() program */
/* --------------------------------------------- */
/* Defining subroutine sec() */
sec(float *a, float *b, float *x1, float *x2, float
     *root, int *count, int *status)
```

```
/*  This subroutine computes a root of an equation using
    the secant method. */
{
     float x3,f1,f2,error;
/*  Function values at initial points */
     *x1 = *a;
     *x2 = *b;
     f1 = F(*a);
     f2 = F(*b);
/*  Compute the root iteratively */
     *count = 1;
     begin:              /* Iteration process begins */
     if(fabs(f1-f2) <= 1.E-10)
     {
        *status = 1;
        return;                  /* Program terminated */
     }
     x3 = *x2 - f2*(*x2-*x1)/(f2-f1);
     error = fabs((x3-*x2)/x3);
/* Test for accuracy */
     if(error > EPS)
     {
/*  Test for convergence */
        if(*count == MAXIT)
        {
           *status = 2;
           return;             /* Program terminated */
        }
        else
        {
           *x1 = *x2;
        }
        *x2 = x3;
        f1 = f2;
        f2 = F(x3);
        *count = *count + 1;
        goto begin;   /* Compute next approximation */
   }
   else
   {
        *root = x3;
        *status = 3;
        return;                    /* Iteration ends */
    }
}
/* End of subroutine sec() */
/* -------------------------------------------------- */
```

## Program 6 FIXEDP

```
/* -------------------------------------------------- *
 * Main program                                        *
 *   This program finds a root of a function using    *
 *   the fixedp point iteration method                 *
 * --------------------------------------------------- *
 * Functions invoked                                   *
 *   Library function fabs() and macro G(x)            *
 * --------------------------------------------------- *
 * Subroutines used                                    *
 *   NIL                                               *
 * --------------------------------------------------- *
 * Variables used                                      *
 *   x0 - Initial guess                                *
 *   x - Estimated root                                *
 *   error - Relative error                            *
 * --------------------------------------------------- *
 *Constants used                                       *
 *   EPS - Error bound                                 *
 *   MAXIT - Maximum iterations allowed                *
 * --------------------------------------------------- */
# include <math.h>
# define EPS  0.000001
# define G(x)  2.0-(x)*(x)

main( )
{
    int MAXIT, i;
    float x0, x, error;
    printf("\n      SOLUTION BY FIXED-POINT METHOD \n");
    printf("\n");
    printf("Input initial estimate of a root \n");
    scanf("%f", &x0);
    printf("Maximum iterations allowed \n");
    scanf("%d", &MAXIT);
    printf("\n   ITERATION   VALUE OF X   ERROR \n");
 /*Iteration process begins*/
    for(i=1;i<=MAXIT;i++)
    {
        x = G(x0);
```

```
      error = fabs((x-x0)/x);
      printf("%10d   %10.7f   %10.7f \n", i, x, error);
      if (error < EPS)
          goto end;    /*Iteration process ends*/
      else
      x0 = x;
    }
    printf("\nProcess does not converge to a root.\n");
    printf("Exit from iteration loop.\n");
    end:
    ;
}
/* End of main() program*/
/* ---------------------------------------------- * /
```

## Program 7 MULTIR

```
/* ---------------------------------------------- *
 * Main program                                     *
 *  The program finds all the real roots of a polynomial *
 * ---------------------------------------------- *
 * Functions invoked                                *
 *  NIL                                             *
 * ---------------------------------------------- *
 * Subroutines used                                 *
 *  newton()                                        *
 *  dflat()                                         *
 * ---------------------------------------------- *
 * Variables used                                   *
 *  n - Degree of polynomial                        *
 *  a - Polynomial coefficients A(N+1)              *
 *  x0 - Initial guess                              *
 *  xr - Root obtained by Newton method             *
 *  root - Root Vector                              *
 *  status - Solution status                        *
 * ---------------------------------------------- *
 * Constants used                                   *
 *  EPS - Error bound                               *
 *  MAXIT - Maximum iterations permitted            *
 * ---------------------------------------------- *

#include <math.h>
#define EPS 0.000001
#define MAXIT 50

void main( )
{
    int n, status, i, j;
    float a[11], root[10], x0, xr;
    void dflat(int n, float a[11], float xr);
    void newton(int n, float a[11], float x0, int *status,
                float *xr);
    printf("\n");
    printf("\n      EVALUATION OF MULTIPLE ROOTS \n");
    printf("\n");
    printf("Input degree of polynomial, n. \n");
    scanf("%d", &n);
    printf("\nInput poly coefficients, a(1) to a(n+1). \n");
    for(i = 1;  i <= n+1; i++)
      scanf("%f",  &a[i]);
    printf("\nInput initial guess of x \n");
    scanf ("%f",  &x0);
    printf("\n");
/* Process of root searching begins */
    for(i=n;i>=2;i--)
    /* Find ith root by Newton Method */
  {
    newton(n,a,x0,&status,&xr);
    if(status == 2)
    {
         for(j=n;j>=i+1;j--)
      printf("root %d = %f \n",j, root[j]);
         printf("\nNext root does not converge in \n");
         printf("%d iterations \n",  MAXIT);
         printf("\n");
         goto end;    /* Processing ends */
    }
     root[i]  =  xr;
/* Deflate the polynomial by division (x-xr) */
    dflat(n,a,xr);
    x0 = xr;
/* Proceed to next root */
  }  /* End of for loop */
/* Compute the last root */
    root[1]  = -a[1]/a[2];
/* Write results */
    printf("\n  ROOTS OF POLYNOMIAL ARE: \n");
    printf("\n");
    for (i=1;i<=n;i++)
      printf("ROOT %d = %f \n",  i, root[i]);
     printf("\n");
     end:
     printf("END");
}
/* End of main() program */
/* --------------------------------------------------* /
/* Defining the subroutine newton() */
void newton(int n, float a[11], float x0, int *status,
              float *xr)
/*  This subroutine finds a root of the polynomial using
    the Newton-Raphson method */
{
      int i, count;
      float fx;  /* Value of polynomial at x0 */
      float fdx; /* Value of polynomial derivative at x0 */
      count = 1;
/* Compute the value of function at x0 */
      begin:
      fx = a[n+1];
      for(i=n;i>=1;i--)
      fx = fx * x0 + a[i];
/* Compute the value of derivative at x0 */
      fdx = a[n+1] * n;
      for(i=n;i>=2;i--)
      fdx = fdx * x0 + a[i] * (i-1);
/* Compute a root xr */
      *xr  =  x0-fx/fdx;
/* Test for accuracy */
      if(fabs((*xr-x0)/(*xr))  <= EPS)
      {
      *status  = 1;
      return;
      }
/*  Test for convergence */
      if(count<MAXIT)
      {
          x0 = *xr;
          count = count + 1;
          goto begin;
      }
      else
      {
          *status = 2;
          return;
        }
}
/* End of subroutine newton() */
/* ------------------------------------------------ */
/* Defining the subroutine dflat() */
void dflat(int n,  float a[11],  float xr)
/*  This subroutine reduces the degree of polynomial by
    one using synthetic division */
{
      float b[11];
      int i;
/* Evaluate the coefficients of the reduced polynomial */
      b[n+1]  = 0;
      for(i=n;i>=1;i--)
      b[i] = a[i+1] + xr * b[i+1];
/* Change coefficients from b array to a array */
      for(i-1;i<=n+1;i++)
      a[i] = b[i];
}
/*  End of subroutine dflat() */
/* ------------------------------------------------ */
```

## Program 8 COMPR

```
* ----------------------------------------------- *
* Main program                                    *
*   This program locates all the roots, both real *
*   and complex, using the Bairstow method        *
* ----------------------------------------------- *
* Functions invoked                               *
*   NIL                                           *
* ----------------------------------------------- *
* Subroutines used                                *
*   input,bstow,quad,output                       *
* ----------------------------------------------- *
* Variables used                                  *
*   n - Degree of polynomial                      *
*   a - Array of coefficients of polynomial       *
*   u0,v0 - Initial values of coefficients of the *
*           quadratic factor                      *
*   u,v   - Computed(values of coefficients of the  *
*           quadratic factor                        *
*   b - Coefficients of the reduced polynomial      *
*   x1,x2 - Roots of the quadratic factor           *
*   type  - Type of roots (real,imaginary or equal) *
* ------------------------------------------------- *
* Constants used                                    *
*   EPS - Error bound                               *
* ------------------------------------------------- */

#include <math.h>
#define  EPS      0.000001
#define  image    1
#define  equal    2
#define  unequal 3

main( )
{
     int n, i;
     float a[11], b[11], u0, v0, u, v, x1, x2, d0, d1,
        d2, root, type, status;

     printf("\n");
     printf("        EVALUATION OF COMPLEX ROOTS \n");
     printf("\n");

/* Get input data */
     printf("Input degree of polynomial, n \n");
     scanf("%d",&n);
     printf("\n Input coefficients a(n+1) to a(1) \n");
     for(i=n+1;i>=1;i--)
        scanf("%f",&a[i]);
     printf("\n Give initial values u0 and v0 \n");
     scanf("%f %f",&u0,&v0);

     begin:
     if(n > 2)
     {
     /* Obtain a quadratic factor */
bstow(n,a[11],b[11],u0,v0,&u,&v,&status);
     if(status == 1)
     {
          d2 = 1;
          d1 = -u;
          d0 = -v;
     }
     else
     {
        printf("\n No Convergence in 100 iterations \n");
        goto end;
     }
     /* Find roots of the quadratic factor */
       quad(d2,d1,d0,&x1,&x2,&type);
     /* Print the roots */
      output(n,type,x1,x2);
     /* Set the coefficients of the factor polynomial */
        n = n-2;
        for(i=1;i<=n+1;i++)
                 a[i] = b[i+2];
     /* Set initial values for next quadratic factor */
        u0 = u;
        v0 = v;
        goto begin;
     }        /* endif */
     if(n == 2)          /* polynomial is quadratic */
     {
        quad(a[3],a[2],a[1],&x1,&x2,&type);
        output(n,type,x1,x2);
     }
     else       /* last root of an odd order polynomial */
     {
        root = -a[1]/a[2];
        printf("\n");
        printf("Final root = %f \n", root);
        printf("\n");
     }
     end:
     printf("End");
}
/* End of main() program */
/* -------------------------------------------------- */
/* Defining the subroutine bstow() */
bstow(int n,float a[11],float b[11],float u0,float v0,
      float *u,float *v,float *status)
/* This subroutine finds the quadratic factor using
   multivariable Newton's method and also finds the
   reduced polynomial */
```

```
{
    float d,delu,delv,c[11];
    int count,i;
    count = 1;
    begin:
    b[n+1] = a[n+1];
    b[n] = a[n] + u0*b[n+1];
    for(i=n-1;i>=1;i--)
        b[i] = a[i] + u0 * b[i+1] + v0 * b[i+2];
    c[n+1] = 0;
    c[n] = b[n+1];
    for(i=n-1;i>=1;i--)
      c[i] = b[i+1] + u0 * c[i+1] + v0 * c[i+2];
    d = c[2] * c[2] - c[1] * c[3];
    delu = -(b[2] * c[2] - b[1] * c[3])/d;
    delv = -(b[1] * c[2] - b[2] * c[2])/d;
    *u = u0 + delu;
    *v = v0 + delv;
    if(fabs(delu/*u) <= EPS && fabs(delv/*v) <= EPS)
    {
      *status = 1;
      return;
    }
    if(count < 100)
    {
      u0 = *u;
      v0 = *v;
      count = count + 1;
      goto begin;
    }
    else
    {
      *status = 2;
      return;
    }
}
/*  End of subroutine bstow() */
/* ----------------------------------------------- */
/*  Define the subroutine quad() */
quad(float a,float b,float c,float *x1,float *x2,float
*type)
```

```
/*  This subroutine solves a quadratic equation of type
    a(x*x) + bx + c */
{
      float q;
      q = b*b - 4*a*c;
      if(q < 0.0)                  /* roots are complex */
    {
      *x1 = -b/(2*a);
      *x2 = sqrt(fabs(q))/(2*a);
      *type = image;
    }
    else if (q == 0.0)   /* roots are real and equal */
    {
      *x1 = -b/(2*a);
      *x2 = *x1;
      *type = equal;
    }
     else                 /* roots are real and unequal */
    {
      *x1 = (-b+sqrt(q))/(2*a);
      *x2 = (-b-sqrt(q))/(2*a);
    }
    return;
}
/* End of subroutine quad() */
/* ------------------------------------------------------- */
/*  Defining output() routine */
output(int n, int type, float x1, float x2)
/* This subroutine displays the roots of the quadratic
    equation */
{
  printf("\n");
  printf("Roots of quadratic factor at n = %d \n", n);
  printf("\n");
  if(type == image)
  {
    printf("Root1 = %f+%fj \n", x1, x2);
    printf("Root2 = %f-%fj \n", x1, x2);
  }
  else if(type == equal)
  {
    printf("Root1 = %f \n", x1);
    printf("Root2 = %f \n", x2);
  }
  else              /* Type == unequal */
  {
    printf("Root1 = %f \n", x1);
    printf("Root2 = %f \n", x2);
  }
  return;
}
/*  End of subroutine output() */
/* ---------------------------------------------------- */
```

## Program 9 MULLER

```
/* ---------------------------------------------------- *
 * Main program                                         *
 *   This program evaluates root of a polynomial using  *
 *   Muller's method                                    *
 * ---------------------------------------------------- *
 * Functions invoked                                    *
 *   NIL                                                *
 * ---------------------------------------------------- *
 * Subroutines used                                     *
 *   F(x)                                               *
 * ---------------------------------------------------- *
 * Variables used                                       *
 *   x1,x2,x3 - initial values                          *
 *   f1,f2,f3 - function values at x1,x2,x3             *
 *   a0,a1,a2 - coefficients of quadratic polynomial    *
 *   hi -   xi-x3                                       *
 *   di -  function difference fi-f3                    *
 * ---------------------------------------------------- *
 * Constants used                                       *
 *   EPS  -  Error bound                                *
 * ---------------------------------------------------- */

#include <math.h>
#define EPS 0.000001

main( )
{
    float F(float x);
    float x1,x2,x3,x4,f1,f2,f3,f4,h1,h2,d1,d2,a0,a1,a2,h;
    printf("\nInput three initial points \n");
    scanf("%f %f %f", &x1,&x2,&x3);

    f1 = F(x1);
    f2 = F(x2);
    f3 = F(x3);

    begin:
    h1 = x1-x3;
```

```
      h2 = x2-x3;
      d1 = f1-f3;
      d2 = f2-f3;
/* Compute parameters a0,a1,a2 */
      a0 = f3;
      a1 = (d2*h1*h1-d1*h2*h2)/(h1*h2*(h1-h2));
      a2 = (d1*h2-d2*h1)/(h*h2*(h1-h2));
/* Compute h */
      if(a1>0.0)
          h=(-2.0*a0)/(a1+sqrt(a1*a1-4*a2*a0));
      else
          h=(-2.0*a0)/(a1-sqrt(a1*a1-4*a2*a0));
/* Compute x4 and f4 */
      x4 = x3+h;
      f4 = F(x4);
/* Test for accuracy */
      if(f4<=EPS) /* root obtained */
      {
          printf("\n\nROOT BY MULLER'S METHOD\n\n");
          printf("Root=%f\n", x4);
          printf("\n");
      }
      else
      {
          x1 = x2;
          x2 = x3;
          x3 = x4;
          f1 = f2;
          f2 = f3;
          f3 = f4;
          goto begin;
      }
}
/* End of main() program */
/* ----------------------------------------------- */
/* Defining the subroutine F(x) */
float F(float x)
{
    float f;
    f = x*x*x+2*x*x+10*x-20;
    return (f);
}
/* End of subroutine () */
/* ----------------------------------------------- */
```

## Program 10 LEG1

```
/ * ---------------------------------------------------- *
  * Main program                                          *
  *   This program solves a system of linear equations *
  *   using simple Gaussian elimination method          *
  * ----------------------------------------------------- *
  * Functions invoked                                     *
  *   NIL                                                 *
  * ----------------------------------------------------- *
  * Subroutines used                                      *
  *   GAUSS1                                              *
  * ----------------------------------------------------- *
  * Variables used                                        *
  *   n - Number of equations in the system               *
  *   a - Matrix of coefficients                          *
  *   b - Right side vector                               *
  *   x - Solution vector                                 *
  * ----------------------------------------------------- *
  *   status - Solution status                            *
  * ----------------------------------------------------- * /

main( )
{
    int status,n,i,j;
    float a[10][10], b[10], x[10];

    printf("\n    SOLUTION BY SIMPLE GAUSS METHOD \n");

    printf("What is the size of the system (n)? \n");
    scanf("%d", &n);

    printf("Input coefficients a(i,j), row-wise, \n");
    printf("one row on each line. \n");
    for(i=1;i<=n;i++)
      for(j=1;j<=n;j++)
          scanf("%f", &a[i][j]);
    printf("\nInput vector b \n");
    for(i=1;i<=n;i++)
      scanf("%f", &b[i]);

/* obtain solution by simple Gauss elimination method */

/* call the subroutine gauss1() */

    gauss1(n,a,b,x,&status);

    if(status != 0)
    {
      printf("\nSOLUTION VECTOR X \n");
```

```
       for(i=1;i<=n;i++)
            printf("%10.6f", x[i]);
       printf("\n");
     }
     else
     {
       printf("Singular matrix, reorder equations. \n");
     }
   }
   /* End of main() program */
   /* ---------------------------------------------------- */
   /* Defining subroutine gauss1() */
   gauss1(int n, float a[10][10], float b[10], float x[10],
          int *status)
   /* This subroutine solves a set of n linear equations
      by Gauss elimination method */
   {
      int i,j,k;
      float pivot, factor, sum;
   /* ---------- Elimination begins ----------- */
      for(k=1;k<=n-1;k++)
      {
        pivot = a[k][k];
        if(pivot < 0.000001)
        {
           *status = 0;
           return;
        }
        *status = 1;
        for(i=k+1;i<=n;i++)
        {
           factor = a[i][k] /pivot;
           for(j=k+1;j<=n;j++)
           {
              a[i][j] = a[i][j] - factor * a[k][j];
           }
           b[i] = b[i] - factor * b[k];
        }
      }
  /* ------ Back substitution begins ------------- */
     x[n] = b[n] / a[n][n];
     for(k=n-1;k>=1;k--)
  {
    sum = 0.0;
    for(j=k+1;j<=n;j++)
        sum = sum + a[k][j] * x[j];
    x[k] = (b[k] - sum) / a[k][k];
  }
  return;
}
/* End of subroutine gauss1 () */
/*-------------------------------------------------*/
```

## Program 11 LEG2

```
/*-------------------------------------------------*
 * Main program                                    *
 *   This program solves a system of linear equations *
 *   using Gaussian elimination with partial pivoting *
 *-------------------------------------------------*
 * Functions invoked                               *
 *   NIL                                           *
 *-------------------------------------------------*
 * Subroutines used                                *
 *   Gauss2                                        *
 *-------------------------------------------------*
 * Variables used                                  *
 *   n - Number of equations                       *
 *   a - Coefficients matrix                       *
 *   b - Right side vector                         *
 *   x - Solution vector                           *
 *-------------------------------------------------*
 * Constants used                                  *
 *   NIL                                           *
 *-------------------------------------------------*/

main( )
{
    int i,j,n;
    float a[10][10], b[10], x[10];

    printf("\n  GAUSS METHOD WITH PARTIAL PIVOTING \n");

    printf("\nWhat is the size n of the system? \n");
    scanf("%d", &n);
    printf("\nInput coefficients a(i,j), row-wise \n");
    printf("one row on each line \n");
    for(i=1;i<=n;i++)
      for(j=1;j<=n;j++)
           scanf("%f",&a[i][j]);
    printf("\nEnter vector b \n");
    for(i=1;i<=n;i++)
    scanf("%f", &b[i]);
    gauss2(n,a,b,x);
    printf("\n          SOLUTION VECTOR X \n");
    printf("\n");
    for(i=1;i<=n;i++)
        printf("\t %f", x[i]);
    printf("\n");
}
/* End of main() program */
/* ------------------------------------------------ */
/* Defining subroutine gauss2() */
gauss2(int n, float a[10][10],float b[10], float x[10])
/* This subroutine solves a system of linear equations using
   Gauss elimination method with partial pivoting */
{
/* Forward elimination */
    elim(n,a,b);
/* Solution by back substitution */
    bsub(n,a,b,x);
    return;
} /* Endof routine gauss2() */
/* ------------------------------------------------ */
/* Defining the subroutine elim() */
[illegible]
/*  This subroutine performs forward elimination
    incorporating partial pivoting technique */
{
      int i,j,k;
      float factor;
      for(k=1;k<=n-1;k++)
      {
        pivot(n,a,b,k);
        for(i=k+1;i<=n;i++)
        {
          factor = a[i][k] / a[k][k];
          for(j=k+1;j<=n;j++)
          {
             a[i][j] = a[i][j]-factor * a[k][j];
          }
          b[i] = b[i]-factor*b[k];
        }
    }
    return;
}   /* End of elim() routine */
/* ------------------------------------------------------ */
/* Defining subroutine pivot() */
# include <math.h>
pivot(int n, float a[10][10], float b[10], int k)
/* This subroutine performs the task of partial pivoting
   (reordering of equations) */
{
      int p,i,j;
      float large, temp;
/* Find pivot p */
      p = k;
      large = fabs(a[k][k]);
      for(i=k+1;i<=n;i++)
      {
          if(fabs(a[i][k])>large)
          {
             large = fabs(a[i][k]);
             p = i;
          }
      }
/* Exchange rows p and k */
      if(p!=k)
      {
          for(j=k;j<=n;j++)
          {
             temp = a[p][j];
             a[p][j] = a[k][j];
             a[k][j] = temp;
          }
          temp = b[p];
          b[p] = b[k];
          b[k] = temp;
      }
      return;
}
/* End of subroutine pivot() */
/* ------------------------------------------------------ */
/* Defining subroutine bsub() */

bsub(int n, float a[10][10], float b[10], float x[10])

/* This subroutine obtains the solution vector x by back
substitution */

{
      int i,j,k;
      float sum;
      x[n] = b[n] / a[n][n];
      for(k=n-1;k>=1;k--)
      {
        sum = 0.0;
        for(j=k+1;j<=n;j++)
             sum = sum + a[k][j] * x[j];
        x[k] = (b[k]-sum) / a[k][k];
      }
      return;
}
/* End of subroutine bsub() */
/* ----------------------------------------------- */
```

## Program 12 DOLIT

```
/* ----------------------------------------------- *
 * Main program                                    *
 *    This program solves a system of linear equations *
 *    using Dolittle LU decomposition              *
 * ----------------------------------------------- *
 * Functions invoked                               *
 *    NIL                                          *
 * ----------------------------------------------- *
 * Subroutines used                                *
 *    LUD, SOLVE                                   *
 * ----------------------------------------------- *
 * Variables used                                  *
 *    n - System size                              *
 *    a - Coefficient matrix of the system         *
 *    b - Right side vector                        *
 *    l - Lower triangular matrix                  *
 *    u - Upper triangular matrix                  *
 *    fact - Factorization status                  *
 * ----------------------------------------------- *
 * Constants used                                  *
 *    YES,NO                                       *
 * ----------------------------------------------- */
```

```
# define YES 1
# define NO  0

main( )
{
    int n, fact, i,j;
    float a[10][10], u[10][10], l[10][10], b[10], x[10];

    printf("\n     SOLUTION BY DOLITTLE METHOD \n\n");

/* Read input data */
    printf("\nWhat is size of A? \n");
    scanf("%d", &n);
    printf("Type coefficients a(i,j), row by row \n");
    for(i=1;i<=n;i++)
       for(j=1;j<=n;j++)
          scanf("%f", &a[i][j]);
      printf("\nType vector b on one line \n");
      for(i=1;i<=n;i++)
          scanf("%f", &b[i]);
/* LU factorization */

     lud(n,a,u,l,&fact);

     if(fact == YES)           /* Print LU matrices */
     {

/* Print U matrix */
     printf("\nMATRIX U \n");
     for(i=1;i<=n;i++)
     {
     for(j=1;j<=n;j++)
     {
       printf("%15.6f", u[i][j]);
     }
     printf("\n");
     }

/* Print L matrix */
     printf("\nMATRIX L \n");
     for(i=1;i<=n;i++)
     {
       for(j=1;j<=n;j++)
       {
          printf("%15.6f", l[i][j]);
       }
       printf("\n");
      }
```

```
/* Solve for x */
      solve(n,u,l,b,x);
      printf("\nSOLUTION VECTOR X \n\n");
      for(i=1;i<=n;i++)
           printf("%15.6f \n", x[i]);
      printf("\n");
      }
      else
      {
      printf("\n       FACTORIZATION NOT POSSIBLE \n");
      }
}
/* End of main() program */
/* ---------------------------------------------------------- */
/* Defining the subroutine lud() */
lud(int n, float a[10][10], float u[10][10], float
l[10][10], int *fact)
/*  This subroutine decomposes the matrix A into L and U
    matrices using DOLITTLE algorithm */
{
        int i,j,k;
        float sum;
/* Initialize U and L matrices */
    for(i=1;i<=n;i++)
      for(j=1;j<=n;j++)
      {
        u[i][j] = 0.0;
        l[i][j] = 0.0;
      }
/* Compute the elements of U and L */
    for(j=1;j<=n;j++)
      u[1][j] = a[1][j];
    for(i=1;i<=n;i++)
      l[i][i] = 1.0;
    for(i=1;i<=n;i++)
       l[i][1] = a[i][1] / u[1][1];

    for(j=2;j<=n;j++)
    {
      for(i=2;i<=j;i++)
      {
        sum = a[i][j];
        for(k=1;k<=i-1;k++)
             sum = sum - l[i][k] * u[k][j];
        u[i][j] = sum;
      }
      if(u[j][j]<=1.E-6)
      {
        *fact = NO;
        return;
      }
      for(i=j+1;i<=n;i++)
      {
        sum = a[i][j];
        for(k=1;k<=j-1;k++)
             sum = sum - l[i][k] * u[k][j];
        l[i][j] = sum / u[j][j];
      }
    }
    *fact = YES;
    return;
} /* End of subroutine lud() */
/* ------------------------------------------------ */
/* Defining the subroutine solve() */
solve(int n, float u[10][10], float l[10][10], float
b[10], float x[10])
/* This subroutine obtains the solution vector X using the
   coefficients of L and U matrices */
{
    int i,j;
    float sum, z[10];
 /* Forward substitution */
     z[1] = b[1];
     for(i=2;i<=n;i++)
     {
       sum = 0.0;
       for(j=1;j<=i-1;j++)
            sum = sum + l[i][j] * z[j];
       z[i] = b[i] - sum;
     }
 /* Back substitution */
       x[n] = z[n]/u[n][n];
       for(i=n-1;i>=1;i--)
       {
         sum = 0.0;
         for(j=i+1;j<=n;j++)
              sum = sum + u[i][j] * x[j];
        x[i] = (z[i]-sum)/u[i][i];
      }
      return;
}
/* End of subroutine solve() */
/* -------------------------------------------------------- */
```

## Program 13 JACIT

```
/* ------------------------------------------------------- *
 *  Main program                                            *
 *     This program uses the subprogram JACOBI to solve     *
 *     a system of equations by Jacobi iteration method     *
 * -------------------------------------------------------- *
 *  Functions invoked                                       *
 *     NIL                                                  *
 * -------------------------------------------------------- *
 *  Subroutines used                                        *
 *     JACOBI                                               *
 * -------------------------------------------------------- *
 *  Variables used                                          *
 *     a - Coefficient matirx                               *
 *     b - Right side vector                                *
 *     n - System size                                      *
 *     x - Solution vector                                  *
 *     count - Number of iterations completed               *
 *     status - Convergence status                          *
 * -------------------------------------------------------- *
 *  Constants used                                          *
 *     EPS - Error bound                                    *
 *     MAXIT - Maximum iterations permitted                 *
 * -------------------------------------------------------- */
#include <math.h>
#define EPS 0.000001
#define MAXIT 100

main( )
{
    int i,j,n,count,status;
    float a[10][10], b[10], x[10];

    printf("\n      SOLUTION BY JACOBI ITERATION \n");
    printf("\nWhat is the size n of the system? \n");
    scanf("%d", &n);
    printf("\nInput coefficients a(i,j), row by row \n");
    for(i=1;i<=n;i++)
      for(j=1;j<=n;j++)
```

```
            scanf("%f", &a[i][j]);
   printf("\nInput vector b\n");
   for(i=1;i<=n;i++)
      scanf("%f", &b[i]);
   jacobi(n,a,b,x, &count, &status);
   if(status==2)
   {
      printf("\nNo convergence in %d iterations", MAXIT);
      printf("\n\n");
   }
   else
   {
      printf("\n                SOLUTION VECTOR X \n\n");
      for(i=1;i<=n;i++)
           printf("%15.6f", x[i]);
      printf("\n\nIterations = %d", count);
   }
}
/* End of main() program */
/* ----------------------------------------------------------- */
/* Defining the routine jacobi() */
jacobi(int n, float a[10][10], float b[10], float x[10], int
*count, int *status)
/*  This subroutine solves a system of n linear equations using
    the Jacobi iteration method */
{
      int i,j,key;
      float sum, x0[10];
/* Initial values of x */
      for(i=1;i<=n;i++)
         x0[i] = b[i] / a[i][i];
/* Jacobi iteration begins */
      *count = 1;
      begin:
      key = 0;
/* Computing values of x(i) */
      for(i=1;i<=n;i++)
      {
         sum = b[i];
         for(j=1;j<=n;j++)
         {
            if(i==j)
                continue;
            sum = sum - a[i][j] * x0[j];
        }
        x[i] = sum / a[i][i];
        if(key == 0)
        {
 /* Testing for accuracy */
            if(fabs((x[i]-x0[i])/x[i]) > EPS)
                key = 1;
        }
    }
 /* Testing for convergence */
        if(key == 1)
        {
          if(*count==MAXIT)
          {
              *status = 2;
              return;
        }
        else
        {
              *status = 1;
              for(i=1;i<=n;i++)
                  x0[i] = x[i];
        }
        *count = *count+1;
        goto begin;
    }
    return;
}
/* End of subroutine jacobi() */
/* ------------------------------------------------------------ */
```

## Program 14 GASIT

```
/* ------------------------------------------------------------
 * Main program                                                *
 *   This program uses the subprogram GASEID to solve a         *
 *   system of equations by Gauss-Seidel iteration method       *
 * ------------------------------------------------------------ *
 * Functions invoked                                            *
 *   NIL                                                        *
 * ------------------------------------------------------------ *
```

```
                                                            *
* Subroutines used                                          *
*    GASEID                                                 *
*-----------------------------------------------------------*
* Variables used                                            *
*    a - Coefficient matrix                                 *
*    b - Right side vector                                  *
*    n - System size                                        *
*    x - Solution vector                                    *
*    count - Number of iterations completed                 *
*    status - Convergence status                            *
*-----------------------------------------------------------*
* Constants used                                            *
*    EPS - Error bound                                      *
*    MAXIT - Maximum iterations permitted                   *
*-----------------------------------------------------------*/

# define MAXIT 50
# define EPS 0.000001

main( )
{
    float a[10][10], b[10], x[10];
    int i,j,n,count,status;
    printf("\n    SOLUTION BY GAUSS-SEIDEL ITERATION \n");
    printf("\nWhat is the size n of the system? \n");
    scanf("%d", &n);
    printf("\nInput coefficients a(i,j), row by row \n");
    printf("one row on each line \n");
    for(i=1;i<=n;i++)
      for(j=1;j<=n;j++)
          scanf("%f", &a[i][j]);
    printf("\nInput vector b\n");
    for(i=1;i<=n;i++)
      scanf("%f", &b[i]);

    gaseid(n,a,b,x, &count, &status);

    if(status==2)
    {
      printf("\nNo convergence in %d iterations \n", MAXIT);
      printf("\n\n");
    }
    else
    {
      printf("\n                  SOLUTION VECTOR X \n\n");
    for(i=1;i<=n;i++)
         printf("%15.6f", x[i]);
    printf("\n\nIterations = %d", count);
```

```
      }
 }
 * End of main() program */
 /* ------------------------------------------------- */
 /*  Defining the routine gaseid() */
 gaseid(int n, float a[10][10], float b[10], float x[10], int
        *count, int *status)
 /*  This subroutine solves a system of linear equations using
     the Gauss-Seidel iteration algorithm */
 {
       int i,j,key;
       float sum, x0[10];
 /* Initial values of x */
       for(i=1;i<=n;i++)
         x0[i] = b[i] / a[i][i];
 /* Gaseid iteration begins */
       *count = 1;
       begin:
       key = 0;
 /* Computing values of x(i) */
       for(i=1;i<=n;i++)
       {
         sum = b[i];
         for(j=1;j<=n;j++)
         {
             if(i==j)
             continue;
             sum = sum - a[i][j] * x0[j];
         }
         x[i] = sum / a[i][i];
         if(key==0)
         {
             /* Testing for accuracy */
             if(fabs((x[i]-x0[i])/x[i])>EPS)
                 key = 1;
         }
       }
 /* Testing for convergence */
      if(key==1)
      {
          if(*count==MAXIT)
          {
            *status = 2;
            return;                  /* Program terminated */
        }
        else
        {
            *status = 1;
            for(i=1;i<=n;i++)
              x0[i] = x[i];
        }
        *count = *count+1;
        goto begin;
      }
      return;
}
* End of routine gaseid() */
/* ---------------------------------------------------------- */
```

## Program 15 LAGRAN

```
/* ---------------------------------------------------------- *
 * -Main program                                               *
 *    This program computes the interpolation value at a      *
 *    specified point, given a set of data points, using      *
 *    the Lagrange interpolation representation               *
 * ---------------------------------------------------------- *
 *  Functions invoked                                          *
 *    NIL                                                      *
 * ---------------------------------------------------------- *
 *  Subroutines used                                           *
 *    NIL                                                      *
 * ---------------------------------------------------------- *
 *  Variables used                                             *
 *    n    - Number of data sets                               *
 *    x(i) - Data points                                       *
 *    f(i) - Function values at data points                    *
 *    xp   - Point at which interpolation is required          *
 *    fp   - Interpolated value at XP                          *
 *    lf   - Lagrangian factor                                 *
 * ---------------------------------------------------------- *
 *  Constants used                                             *
 *    MAX - Maximum number of data points permitted            *
 * ---------------------------------------------------------- *

#define MAX 10

main( )
{
      int n,i,j;
      float x[MAX],f[MAX],fp,lf,sum,xp;

      printf("\nInput number of data points, n \n");
      scanf("%d", &n);

      printf("\nInput data points x(i) and values f(i) \n");
      printf("(one set in each line)\n")
      for(i=1;i<=n;i++)
           scanf("%f %f", &x[i], &f[i]);

      printf("\nInput x at which interpolation
        is required. \n");
      scanf("%f", &xp);

      sum = 0.0;
      for(i=1;i<=n;i++)
      {
        lf = 1.0;
        for(j=1;j<=n;j++)
        {
              if(i!=j)
                 lf = lf*(xp-x[j])/(x[i]-x[j]);
        }
        sum = sum + lf * f[i];
      }
      fp = sum;
      printf("\n       LAGRANGIAN INTERPOLATION \n\n");

      printf("Interpolated function value \n");
      printf("at x=%f is %f \n", xp, fp);
}
/* End of main() program */

/* ------------------------------------------------- */
```

## Program 16 NEWINT

```
/* ------------------------------------------------ *
 * Main program                                      *
 *    This program constructs the Newton interpolation *
 *    polynomial for a given set of data points and then *
 *    computes interpolation value at a specified value *
 * ------------------------------------------------ *
 * Functions invoked                                 *
 *    NIL                                            *
 * ------------------------------------------------ *
 * Subroutines used                                  *
 *    NIL                                            *
 * ------------------------------------------------ *
 *                                                            *
 * Variables used                                             *
 *    n  - Number of data points                              *
 *    x  - Array of independent data points                   *
 *    f  - Array of function values                           *
 *    xp - Desired point for interpolation                    *
 *    fp - Interpolation value at XP                          *
 *    f  - Difference table                                   *
 *    a  - Array of coefficients of interpolation             *
 *         polynomial                                         *
 * ---------------------------------------------------------- *
 * Constants used                                             *
 *   NIL                                                      *
 * ---------------------------------------------------------- */

main( )
{
      int i,j,n;
      float xp, fp, sum, pi, x[10], f[10], a[10], d[10][10];

      printf("\nInput number of data points \n");
      scanf("%d", &n);
      printf("\nInput values of x and f(x), \n");
      printf("One set on each line \n");

      for(i=1;i<=n;i++)
      scanf("%f %f", &x[i], &f[i]);

/* Construct difference table */
      for(i=1;i<=n;i++)
        d[i][1] = f[i];

      for(j=2;j<=n;j++)
        for(i=1;i<=n-j+1;i++)
             d[i][j] = (d[i+1][j-1]-d[i][j-1])/(x[i+j-1]-x[i]);

/* Set the coefficients of interpolation polynomial */
      for(j=1;j<=n;j++)
         a[j] = d[1][j];

/* Compute interpolation value */
      printf("\nInput xp where interpolation is required \n");
      scanf("%f", &xp);
      sum = a[1];
      for(i=2;i<=n;i++)
      {
        pi = 1.0;
        for(j=1;j<=i-1;j++)
             pi = pi * (xp-x[j]);
        sum = sum + a[i] * pi;
      }
      fp = sum;
/* Write results */
      printf("\n");
      printf("     NEWTON INTERPOLATION \n");
      printf("\n");
      printf("Interpolated value \n");
      printf("at x = %f is %f \n", xp, fp);
}
/* End of main() program */
/* ------------------------------------------------ */
```

## Program 17 SPLINE

```
/* ------------------------------------------------ *
 *  Main program                                     *
 *    This program computes the interpolation value at *
 *    a specified value, given a set of table points, *
 *    using the natural cubic spline interpolation   *
 * ------------------------------------------------ *
 *  Functions invoked                                *
 *    NIL                                            *
 * ------------------------------------------------ *
 *  Subroutines used                                 *
 *    GAUSS                                          *
 * ------------------------------------------------ *
 *  Variables used                                   *
 *    n  - Number of data points.                    *
 *    x  - N by 1 array of data points.              *
 *    f  - N by 1 array of function values           *
 *    xp - Point at which interpolation is required  *
 *    fp - Interpolation value at XP                 *
 *    a  - Array of second derivatives (N-2 by 1)    *
 *    d  - Array representing (N-2 by 1)             *
 *    c  - Matrix (N-2 by N-2) representing the      *
 *         coefficients of second derivatives        *
 *    h  - Array of distances between data points    *
 *         ( h(i) = x(i) - x(i-1) )                  *
 *    df - Array of differences of functions         *
 * ------------------------------------------------ *
 *  Constants used                                   *
 *    MAX - Maximum number of table points permitted *
 * ------------------------------------------------ */
#define S 10
main( )
{
      int i,j,n,m;
      float x[S],f[S],a[S],d[S],c[S][S],h[S],df[S],u[S],
            xp,fp,q1,q2,q3;
/* Reading input data */
      printf("\nInput number of data points, n \n");
      scanf("%d", &n);
      printf("\nInput data points x(i) and function \n");
      printf("values f(i), one set on each line \n");
      for(i=1;i<=n;i++)
        scanf("%f %f", &x[i],&f[i]);
      printf("\nInput point of interpolation \n");
      scanf("%f", &xp);
/* Compute distances between data points and function
   differences */
      for(i=2;i<=n;i++)
      {
        h[i] = x[i] - x[i-1];
        df[i] = f[i] - f[i-1];
      }
/* Initialize C matrix */
      for(i=2;i<=n-1;i++)
        for(j=2;j<=n-1;j++)
             c[i][j] = 0.0;
/* Compute diagonal elements of c */
      for(i=2;i<=n-1;i++)
        c[i][i] = 2.0 * (h[i] - h[i+1]);
/* Compute off-diagonal elements of c */
      for(i=3;i<=n-1;i++)
      {
        c[i-1][i] = h[i];
        c[i][i-1] = h[i];
      }
/* Compute elements of d array */
      for(i=2;i<=n-1;i++)
        d[i] = (df[i+1]/h[i+1] - df[i]/h[i]) * 6.0;
/* Compute elements of a using Gaussian elimination.
   Change array subscripts from 2 to n-1 to 1 to n-2
   before calling gauss() subroutine. */
      m = n-2;
      for(i=1;i<=m;i++)
      {
        d[i] = d[i+1];
        for(j=1;j<=m;j++)
            c[i][j] = c[i+1][j+1];
      }
      gauss(m,c,d,a);
/*  Compute the coefficients of natural cubic spline */
      for(i=n-1;i>=2;i--)
        a[i] = a[i-1];
      a[1] = 0.0;
      a[n] = 0.0;
/*  Locate the domain of xp */
      for(i=2;i<=n;i++)
      {
        if(xp <= x[i])
            break;
      }
/* Compute interpolation value at xp */
      u[i-1] = xp - x[i-1];
      u[i] = xp - x[i];
      q1 = h[i]*h[i] * u[i] - u[i]*u[i]*u[i];
      q2 = u[i-1]*u[i-1]*u[i-1] - h[i]*h[i] * u[i-1];
      q3 = f[i]*u[i-1] - f[i-1]*u[i];
      fp = (a[i-1]*q1 + a[i]*q2)/(6.0 * h[i]) + q3/h[i];
/* Write results */
      printf("\n          SPLINE INTERPOLATION \n \n");
      printf("Interpolation value = %f \n", fp);
}
/* End of main() program */
/* ---------------------------------------------- */
/*  Defining gauss() subroutine */
gauss(int n, float a[10][10], float b[10], float x[10])
/* This subroutine solves a set of n linear equations using
   Gauss elimination method */
{
      int i,j,k;
      float pivot, factor, sum;
/* ---------- Elimination begins ----------- */
      for(k=1;k<=n-1;k++)
      {
        pivot = a[k][k];
        for(i=k+1;i<=n;i++)
        {
          factor = a[i][k] / pivot;
          for(j=k+1;j<=n;j++)
            a[i][j] = a[i][j] - factor * a[k][j];
          b[i] = b[i] - factor * b[k];
        }
      }
/* Back substitution begins */
      x[n] = b[n]/a[n][n];
      for(k=n-1;k>=1;k–)
      {
        sum = 0.0;
        for(j=k+1;j<=n;j++)
            sum = sum + a[k][j] * x[j];
        x[k] = (b[k]-sum)/a[k][k];
      }
      return;
}
/* End of subroutine gauss() */
/* ---------------------------------------------- */
```

## Program 18 LINREG

```
/*----------------------------------------------- *
 * Main program                                   *
 *   This program fits a line Y = A + BX  to a given *
 *   set of data points by the method least squares *
 *----------------------------------------------- *
 * Functions invoked                              *
 *   Library function fabs()                      *
 *----------------------------------------------- *
 * Subroutines used                               *
 *   NIL                                          *
 *----------------------------------------------- *
 * Variables used                                 *
 *   x,y - Data arrays                            *
 *   n - Number of data sets                      *
 *   sumx - Sum of x values                       *
 *   sumy - Sum of y values                       *
 *   sumxx - Sum of squares of x values           *
 *   sumxy - Sum of products of x and y           *
 *   xmean - Mean of x values                     *
 *   ymean - Mean of y values                     *
 *   a - y intercept of the line                  *
 *   b - Slope of the line                        *
 *----------------------------------------------- *
 *  Constants used                                          *
 *     MAX - Limit for number of data points                *
 *     EPS - Error limit                                    *
 * -------------------------------------------------------- *

#include <math.h>
#define MAX 10
#define EPS 0.000001

main( )
{
      int i,n;
      float x[10], y[10];
      float sumx, sumy, sumxx, sumxy, xmean, ymean,
            denom, a, b;

      printf("\n      LINEAR REGRESSION \n \n");

/* Reading data values */
      printf("\nInput number of data points, n \n");
      scanf("%d", &n);
      printf("\nInput x and y values \n");
      printf("One set on each line \n");
      for(i=1;i<=n;i++)
        scanf("%f %f", &x[i], &y[i]);

/* Computing constants a and b of the linear equation */
      sumx = 0.0;
      sumy = 0.0;
      sumxx = 0.0;
      sumxy = 0.0;
      for(i=1;i<=n;i++)
      {
        sumx = sumx + x[i];
        sumy = sumy + y[i];
        sumxx = sumxx + x[i] * x[i];
        sumxy = sumxy + x[i] * y[i];
      }
      xmean = sumx/n;
      ymean = sumy/n;
      denom = n*sumxx - sumx * sumx;
      if(fabs(denom) > EPS)
      {
        b = (n*sumxy - sumx * sumy)/denom;
        a = ymean - b * xmean;
        printf("\n   LINEAR REGRESSION LINE y = a+bx \n");

        printf("\nThe coefficients are: \n");
        printf("a = %f \n", a);
        printf("b = %f \n", b);
      }
      else
      {
        printf("\n NO SOLUTION \n");
      }
}
/*  End of main() program */
/* --------------------------------------------------*/
```

## Program 19 POLREG

```
/* -------------------------------------------------- *
 *  Main program                                       *
 *    This program fits a polynomial curve to a given  *
 *    set of data points by the method of              *
 *    least squares                                    *
 * -------------------------------------------------- *
 *  Functions invoked                                  *
 *    NIL                                              *
 * -------------------------------------------------- *
 *  Subroutines used                                   *
 *    normal, gauss                                    *
 * -------------------------------------------------- *
 *  Variables used                                     *
 *    x,y - Arrays of data values                      *
 *    n - Number of data points                        *
 *    mp - Order of the polynomial under construction  *
 *    m - Number of polynomial coefficients            *
 *    c - Coefficient matrix of normal equations       *
 *    b - Right side vector of normal equations        *
 *    a - Array of coefficients of the polynomial      *
 * -------------------------------------------------- *
 *  Constants used                                     *
 *    MAX - Maximum number of data points              *
 * -------------------------------------------------- */

#include <math.h>
#define MAX 10

main( )
{
      int n,mp,m,i;
      float x[MAX],y[MAX],c[MAX][MAX],a[MAX],b[MAX];

      printf("\n      POLYNOMIAL REGRESSION \n\n");
```

```
/* Reading values */
     printf("Input number of data points n \n");
     scanf("%d",&n);
     printf("Input order of polynomial, mp required \n");
     scanf("%d",&mp);
     printf("Input data values, x and y \n");
     printf("one set on each line \n");
     for(i=1;i<=n;i++)
       scanf("%f %f", &x[i], &y[i]);

/* Testing the order */
     if(n <= mp)
     {
       printf("\n  REGRESSION IS NOT POSSIBLE \n");
       goto stop;
     }

/* Number of polynomial coefficients */
     m = mp + 1;

/* Computation of elements of c and b */
     normal(x,y,c,b,n,m);

/* Computation of coefficients a(1) to a(m) */
     gauss(m,c,b,a);

/* Printing of coefficients a(i) */
     printf("\n    POLYNOMIAL COEFFICIENTS \n\n");
     for(i=1;i<=m;i++)
       printf("%15.6f", a[i]);
     printf("\n");
     stop:
     printf("END");

}
/* End of main() program */
/* ------------------------------------------------------*/

/* Defining the subroutine normal() */
normal(float x[MAX], float y[MAX], float c[MAX][MAX],
      float b[MAX], int n, int m)

/* This subroutine computes the coefficients of normal
   equations */
{
     int i,j,k,l1,l2;
     for(j=1;j<=m;j++)
     {
       for(k=1;k<=m;k++)
       {
          c[j][k] = 0.0;
          11 = k+j-2;
          for(i=1;i<=n;i++)
            c[j][k] = c[j][k] + pow(x[i],11);
       }
     }
     for(j=1;j<=m;j++)
     {
       b[j] = 0.0;
       12 = j-1;
       for(i=1;i<=n;i++)
           b[j] = b[j]+y[i] * pow(x[i],12);
     }
     return;
}
/* End of subroutine normal() */
/* ---------------------------------------------- */
/* Defining gauss() subroutine */
gauss(int n, float a[MAX][MAX], float b[MAX], float x[MAX])
/* This subroutine solves a set of n linear equations by Gauss
   Elimination method */
{
     int i,j,k;
     float pivot, factor, sum;
/* Elimination begins */
     for(k=1;k<=n-1;k++)
     {
        pivot = a[k][k];
        for(i=k+1;i<=n;i++)
        {
           factor = a[i][k]/pivot;
           for(j=k+1;j<=n;j++)
               a[i][j] = a[i][j] - factor * a[k][j];
           b[i] = b[i] - factor * b[k];
        }
     }
/* Back substitution begins */
     x[n] = b[n]/a[n][n];
     for(k=n-1;k>=1;k--)
     {
       sum = 0.0;
       for(j=k+1;j<=n;j++)
```

```
            sum = sum + a[k][j] * x[j];
        x[k] = (b[k]-sum)/a[k][k];
      }
      return;
}
/* End of subroutine gauss() */
/* -------------------------------------------------------- */
```

## Program 20 NUDIF

```
/* ------------------------------------------------------- *
 *  Main program                                            *
 *    This program computes the derivative of a tabulated   *
 *    function at a specified value using the Newton*
 *    interpolation approach                               *
 * ------------------------------------------------------- *
 *  Functions invoked                                      *
 *    NIL                                                  *
 * ------------------------------------------------------- *
 *  Subroutines used                                       *
 *    NIL                                                  *
 * ------------------------------------------------------- *
 *  Variables used                                         *
 *    n - Number of function values given                  *
 *    x - Array of values                                  *
 *    f - Array of function values                         *
 *    a - Array of coefficients of Newton polynomial       *
 *    d - Difference table                                 *
 *    xp - Desired point of differentiation                *
 *    dif - Derivative of the function at XP               *
 * ------------------------------------------------------- *
 *  Constants used                                         *
 *    NIL                                                  *
 * ------------------------------------------------------- */

main( )
{
      int i,j,k,n;
      float x[10],f[10],a[10],d[10][10],xp,dif,sum,p;

/* Reading input data */
      printf("\nInput number of data points \n");
      scanf("%d", &n);
      printf("\nInput values of x and f(x), \n");
      printf("one set on each line \n");
      for(i=1;i<=n;i++)
        scanf("%f %f", &x[i], &f[i]);
/* Constructing difference table d */
     for(i=1;i<=n;i++)
       d[i][1] = f[i];
     for(j=2;j<=n;j++)
       for(i=1;i<=n-j+1;i++)
            d[i][j] = (d[i+1][j-1] - d[i][j-1])/(x[i+j-1] -
x[i]);
/* Set the coefficients of Newton interpolating polynomial */
     for(j=1;j<=n;j++)
       a[j] = d[1][j];
/* Compute derivative at a given x = xp */
     printf("\nInput xp where derivative is required \n");
     scanf("%f", &xp);
     dif = a[2];
     for(k=3;k<=n;k++)
     {
       sum = 0.0;
       for(i=1;i<=k-1;i++)
       {
           p = 1.0;
           for(j=1;j<=k-1;j++)
           {
             if(i == j)
                continue;
             p = p * (xp - x[j]);
          }
          sum = sum + p;
       }
       dif = dif + a[k] * sum;
     }
/* Write results */
     printf("\nNUMERICAL DIFFERENTIATION USING ");
     printf("NEWTON POLYNOMIAL \n\n");
     printf("DERIVATIVE at x = %f is %f \n", xp,dif);
}
/* End of main() program */
/* -------------------------------------------- */
```

## Program 21 TRAPE1

```
/* ------------------------------------------- *
 * Main program                                *
 *   This program integrates a given function  *
 *   using the trapezoidal rule                *
 * ------------------------------------------- *
```

```
                                                              *
 *  Functions invoked                                            *
 *     NIL                                                       *
 * ------------------------------------------------------------- *
 *  Subroutines used                                             *
 *     F(x)                                                      *
 * ------------------------------------------------------------- *
 *  Variables used                                               *
 *       a - Lower limit of integration                          *
 *       b - Upper limit of integration                          *
 *       h - Segment width                                       *
 *       n - Number of segments                                  *
 *     ict - Value of integral                                   *
 * ------------------------------------------------------------- *
 *  Constants used                                               *
 *     NIL                                                       *
 * ------------------------------------------------------------- */

#include <math.h>

Main( )
{
      int n,i;
      float a,b,h,sum,ict;
      float F(float x):

      printf("\nGive initial value of x \n");
      scanf("%f",&a);
      printf("\nGive final value of x \n");
      scanf("%f", &b);
      printf("\nWhat is the segment width? \n");
      scanf("%f", &h);

      n = (b-a)/h;

      sum = (F(a) + F(b))/2.0;
      for(i=1;i<=n-1;i++)
      {
        sum = sum + F(a+i*h);
      }

      ict = sum * h;

      printf("\n");
      printf("Integration between %f and %f \n", a,b);
      printf("When h = %f is %f \n", h, ict);
      printf("\n");
}
/* End of main() program */
/* ------------------------------------------------------------- */
```

```
float F(float x)
{
      float f;
      f = 1.0-exp(-x/2.0);
      return (f);
}
/* End of subroutine F(x) */
/* ---------------------------------------------------- */
```

## Program 22 SIMS1

```
/* ---------------------------------------------------- *
 * Main program                                         *
 *   This program integrates a given function           *
 *   using the Simpson's 1/3 rule                       *
 * ---------------------------------------------------- *
 * Functions invoked                                    *
 *   Macro F(x)                                         *
 * ---------------------------------------------------- *
 * Subroutines used                                     *
 *   NIL                                                *
 * ---------------------------------------------------- *
 * Variables used                                       *
 *   a - Lower limit of integration                     *
 *   b - Upper limit of integration                     *
 *   h - Segment width                                  *
 *   n - Number of segments                             *
 * ics - Value of the integral                          *
 * ---------------------------------------------------- *
 * Constants used                                       *
 *   NIL                                                *
 * ---------------------------------------------------- */
#include <math.h>
#define F(x) 1 - exp(-(x)/2.0)
main( )
{
      int n,m,i;
      float a,b,h,sum,ics,x,f1,f2,f3;
      printf("\nInitial value of x \n");
      scanf("%f", &a);
      printf("\nFinal value of x \n");
      scanf("%f", &b);
      printf("\nNumber of segments (EVEN number) \n");
      scanf("%d", &n);
      h = (b-a)/n;
      m = n/2;
      sum = 0.0;
      x = a;
      f1 = F(x);
      for(i=1;i<=m;i++)
      {
        f2 = F(x+h);
        f3 = F(x+2*h);
        sum = sum + f1 + 4*f2 + f3;
        f1 = f3;
        x = x + 2*h;
      }
      ics = sum * h/3.0;
      printf("\nIntegral from %f to %f \n", a,b);
      printf("When h = %f is %f \n", h, ics);
}
/*  End of main() program */
--------------------------------------------------------- */
/* -------------------------------------------------------
```

## Program 23 ROMBRG

```
--------------------------------------------------------- *
/ * ----------------------------------------------------- *
 *  Main program                                          *
 *    This program performs Romberg integration           *
 *    by bisecting the intervals N times                  *
 * ------------------------------------------------------ *
 *  Functions invoked                                     *
 *    Macro F(x), Library function fabs()                 *
 * ------------------------------------------------------ *
 *  Subroutines used                                      *
 *    NIL                                                 *
 * ------------------------------------------------------ *
 *  Variables used                                        *
 *    a - Starting point of the interval                  *
 *    b - End point of the interval                       *
 *    h - Width of the interval                           *
 *    n - Number of times bisection is done               *
 *    m - Number of trapezoids                            *
 *    r - Matrix of Romberg integral values               *
 * ------------------------------------------------------ *
 *  Constants used                                        *
 *    EPS - Error bound                                   *
 * ------------------------------------------------------ */
```

```
#include <math.h>
#define EPS 0.000001
#define F(x) 1.0/(x)
main( )
{
      int i,j,k,m,n;
      float a,b,h,sum,x,r[10][10];
      printf("\nInput end points of the interval \n");
      scanf("%f %f", &a,&b);
      printf("\nInput maximum number of times");
      printf("\nthe subintervals are bisected \n");
      scanf("%d", &n);
/* Compute area using entire interval as one trapezoidal */
      h = b - a;
      r[1][1] = h * (F(a)+F(b))/2.0;
      printf("\n%15.6f \n", r[1][1]);
/* Process of Romberg integration begins */
      for(i=2;i<=n+1;i++)
      {
        m = pow(2,(i-2)); /* trapezoidal for ith refinement */
        h = h/2;          /* bisect step-size */
/* Use recursive trapezoidal rule for m strips */
        sum = 0.0;
        for(k=1;k<=m;k++)
        {
          x = a + (2*k-1)*h;
          sum = sum + F(x);
        }
        r[i][1] = r[i-1][1]/2.0 + h*sum;
/* Compute Richardson's improvements */
      for(j=2;j<=i;j++)
      {
         r[i][j]=r[i][j-1]+(r[i][j-1]-r[i-1][j-1])/(pow(4,j-1)-1);
      }
/* Write results of improvements for ith refinement */
      for(j=1;j<=i;j++)
          printf("%15.6f", r[i][j]);
      printf("\n");
/* Test for accuracy */
      if(fabs(r[i-1][i-1] - r[i][i]) < EPS)
      /* Stop further refinement */
      {
         printf("\n");
         printf("ROMBERG INTEGRATION = %f \n", r[i][i]);
         printf("\n");
         goto stop;
       }
       else
       {
         continue;
       }
     }
/* Write final result */
     printf("\nROMBERG INTEGRATION = %f \n", r[n+1][n+1]);
     printf("(Normal exit from loop) \n");

     stop:
     printf("End");
}
/*  End of main() program */
/* ----------------------------------------------------- */
```

## Program 24 TRAPE2

```
/* ---------------------------------------------------- *
 *  Main program                                         *
 *    This program integrates a tabulated function      *
 *    using the trapezoidal rule                         *
 * ---------------------------------------------------- *
 *  Functions invoked                                    *
 *    Library function fabs()                            *
 * ---------------------------------------------------- *
 *  Subroutines used                                     *
 *    NIL                                                *
 * ---------------------------------------------------- *
 *  Variables used                                       *
 *      n - Number of table points                       *
 *      x - Array of independent data points             *
 *      y - Array of function values                     *
 *      a - Lower limit of integration                   *
 *      b - Upper limit of integration                   *
 *      h - Distance between points                      *
 *     n1 - Position of A in the table                   *
 *     n2 - Position of B in the table                   *
 *    ict - Value of integral                            *
 * ---------------------------------------------------- *
 *  Constants used                                       *
 *    NIL                                                *
 * ---------------------------------------------------- */
```

```
#include <math.h>
#define MAX 15

main( )
{
      int n,n1,n2,i;
      float  a,b,h,sum,ict,x[MAX],y[MAX];

/* Reading table values */
      printf("Number of data points \n");
      scanf("%d", &n);
      printf("\nInput table values, set by set \n");
      for(i=1;i<=n;i++)
      scanf("%f %f", &x[i],&y[i]);

/* Reading the limits of integration */
      printf("Initial value of x \n");
      scanf("%f",&a);
      printf("Final value of x \n");
      scanf("%f", &b);
      printf("\nWhat is segment width? \n");
      scanf("%f", &h);

/* Computing the position of initial and final values */
      n1 = (int)(fabs(a-x[1])/h+1.5);
      n2 = (int)(fabs(b-x[1])/h+1.5);

/* Evaluating the integral */
      sum = 0.0;
      for(i=n1;i<=n2-1;i++)
        sum = sum + y[i] + y[i+1];
      ict = sum * h/2.0;
      printf("\nIntegral from %f to %f is %f\n", a,b,ict);
}
/*  End of main() program */
/* ------------------------------------------------------ * /
```

## Program 25 SIMS2

```
/ * ---------------------------------------------------- *
 *  Main program                                         *
 *    This program integrates a tabulated function       *
 *    using the Simpson's 1/3 rule.If the number of      *
 *    segments is odd, the trapezoidal rule is used      *
 *    for the last segment.                              *
 * ----------------------------------------------------- *
 *  Functions invoked                                    *
 *    Library function fabs()                            *
 * ----------------------------------------------------- *
 *  Subroutines used                                                  *
 *     NIL                                                            *
 * ------------------------------------------------------------------ *
 *  Variables used                                                    *
 *      n - Number of table points                                    *
 *      x - Array of independent values                               *
 *      y - Array of function values                                  *
 *      a - Lower limit of integration                                *
 *      b - Upper limit of integration                                *
 *      h - Distance between independent values                       *
 *     n1 - Position of A in the table                                *
 *     n2 - Position of B in the table                                *
 *     i1 - Area computed using Simpson's rule                        *
 *     i2 - Area of the last segment (by trapezoidal rule)            *
 *    ics - Value of integral                                         *
 * ------------------------------------------------------------------ *
 *  Constants used                                                    *
 *     NIL                                                            *
 * ------------------------------------------------------------------ */

#include <math.h>
main( )
{
      int i,n,n1,n2,m,l;
      float x[15],y[15],a,b,h,sum,ics,i1,i2;

/* Reading table values */
      printf("Input number of data points \n");
      scanf("\%d", &n);
      printf("\nInput table values, set by set \n");
      for(i=1;i<=n;i++)
        scanf("%f %f", &x[i],&y[i]);

/* Reading the limits of integration */
      printf("Input initial value of x \n");
      scanf("%f", &a);
      printf("Input final value of x \n");
      scanf("%f", &b);
      printf("What is segment width? \n");
      scanf("%f", &h);

/* Computing the position of initial and final values */
      n1 = (int)(fabs(a-x[1])/h+1.5);
      n2 = (int)(fabs(b-x[1])/h+1.5);

/* Testing for even intervals */
      m = n2 - n1;
      if(m%2 == 0) /* m is even */
      {
        i2 = 0.0;
        l = n2 - 2;
      }
      else /* m is odd */
      {
      /* Use trapezoidal rule for the last strip */
        i2 = (y[n2-1]+y[n2])*h/2.0;
        l = n2 - 3;
      }
/* Use Simpson's rule for l strips */
      sum = 0.0;
      for(i=n1;i<=l;i=i+2)
        sum = sum + y[i] + 4*y[i+1] + y[i+2];
      i1 = sum*h/3.0;
/* Integral is sum of i1 and i2 */
      ics = i1 + i2;
/* Writing the results */
      printf("\nIntegral from %f to %f is %f \n", a,b,ics);
}
/*  End of main() program */
/* -------------------------------------------------- */
```

### Program 26 EULER

```
/* ------------------------------------------------ *
 *                                                  *
 * Main program                                     *
 *   This program  estimates  the solution of the first *
 *   order differential equation y' = f(x,y) at a given *
 *   point using Euler's method                     *
 * ------------------------------------------------ *
 * Functions invoked                                *
 *   NIL                                            *
 * ------------------------------------------------ *
 * Subroutines used                                 *
 *   func()                                         *
 * ------------------------------------------------ *
 * Variables used                                   *
 *      x - Initial value of independent variable   *
 *      y - Initial value of dependent variable     *
 *     xp - Point of solution                       *
 *      h - Incremental step-size                   *
 *      n - Number of computational steps required  *
 *     dy - Incremental Y in each step              *
 * ------------------------------------------------ *
```

```
  * Constants used                                           *
  *    NIL                                                   *
  * ------------------------------------------------------ * /

main( )
{
      int i,n;
      float x,y,xp,h,dy;
      float func(float, float);

      printf("\n     SOLUTION BY EULER'S METHOD \n \n");

/* Reading initial data                                     */
      printf("Input initial values of x and y \n");
      scanf("%f %f", &x,&y);
      printf("Input x at which y is required \n");
      scanf("%f", &xp);
      printf("Input step-size, h \n");
      scanf("%f", &h);

/* Compute number of steps required */
      n = (int)((xp-x)/h+0.5);

/* Compute y recursively at each step */
      for(i=1;i<=n;i++)
      {
        dy = h * func(x,y);
        x = x + h;
        y = y + dy;
        printf("%5d %10.6f %10.6f \n", i,x,y);
      }

/* Write the final result */
      printf("\nValue of y at x = %f is %f \n", x,y);
}
/*  End of main() program */

/* ------------------------------------------------------- * /

/* Defining subroutine func() */

float func(float x, float y)

{
      float f;
      f = 2.0 * y/x;
      return(f);
}
/*  End of subroutine func() */

/* ------------------------------------------------------- * /
```

## Program 27 HEUN

```
/* ------------------------------------------------------------ *
 *  Main program                                                *
 *    This program solves the first order differential          *
 *    equation y' = f(x,y) using the Heun's method              *
 * ------------------------------------------------------------ *
 *  Functions invoked                                           *
 *    NIL                                                       *
 * ------------------------------------------------------------ *
 *  Subroutines used                                            *
 *    func()                                                    *
 * ------------------------------------------------------------ *
 *  Variables used                                              *
 *    x  - Initial value of independent variable                *
 *    y  - Initial value of dependent variable                  *
 *    xp - Point of solution                                    *
 *    h  - Step-size                                            *
 *    n  - Number of steps                                      *
 * ------------------------------------------------------------ *
 *  Constants used                                              *
 *    NIL                                                       *
 * ------------------------------------------------------------ */

main( )
{
      int i,n;
      float x,y,xp,h,m1,m2;
      float func(float, float);

      printf("\n   SOLUTION BY HEUN'S METHOD \n \n");

/* Reading initial data */
      printf("Input initial values of x and y \n");
      scanf("%f %f", &x,&y);
      printf("Input x at which y is required \n");
      scanf("%f", &xp);
      printf("Input step-size, h \n");
      scanf("%f", &h);

/* Compute number of steps required */
      n = (int)((xp-x)/h+0.5);

/* Compute y recursively at each step */
      for(i=1;i<=n;i++)
      {
        m1 = func(x,y);
        m2 = func(x+h,y+m1*h);
        x = x + h;
        y = y + 0.5*h*(m1+m2);
        printf("%5d %10.6f %10.6f \n", i,x,y);
      }
/* Write the final result */
      printf("\nValue of y at x = %f is %f \n", x,y);
}
/* End of main() program */
/* ------------------------------------------------ */
/* Defining subroutine func() */
float func(float x, float y)
{
      float f;
      f = 2.0 * y/x;
      return(f);
}
/* End of subroutine func() */
/* ------------------------------------------------ */
```

## Program 28 POLYGN

```
/* ----------------------------------------------- *
 *                                                  *
 * Main program                                     *
 *   This program solves the differential equation *
 *   of type y' = f(x,y) by polygon method          *
 * ----------------------------------------------- *
 *                                                  *
 * Functions invoked                                *
 *   NIL                                            *
 * ----------------------------------------------- *
 *                                                  *
 * Subroutines used                                 *
 *   func()                                         *
 * ----------------------------------------------- *
 *                                                  *
 * Variables used                                   *
 *   x - Initial value of the independent variable *
 *   y - Initial value of the dependent variable   *
 *   xp - Point of solution                         *
 *   h - Incremental step-size                      *
 *   n - Number of computational steps required    *
 * ----------------------------------------------- *
 *                                                  *
 * Constants used                                   *
 *   NIL                                            *
 * ----------------------------------------------- */

main( )
{
      int i,n;
      float x,y,xp,h,m1,m2;
      float func(float, float);
      printf("\n  SOLUTION BY POLYGON METHOD \n \n");
/* Reading initial data */
      printf("Input initial values of x and y \n");
      scanf("%f %f", &x,&y);
      printf("Input x at which y is required \n");
      scanf("%f", &xp);
      printf("Input step-size, h \n");
      scanf("%f", &h);
/* Compute number of steps required */
      n = (int)((xp-x)/h+0.5);
/* Compute y recursively at each step */
      for(i=1;i<=n;i++)
      {
        m1 = func(x,y);
        m2 = func(x+0.5*h,y+0.5*h*m1);
        x = x + h;
        y = y + m2*h;
        printf("%5d %10.6f %10.6f \n", i,x,y);

      }
/* Write the final result */
      printf("\nValue of y at x = %f is %f \n", x,y);

}
/*  End of main() program */
/* ----------------------------------------------- */

/*  Defining subroutine func() */
float func(float x, float y)
{
      float f;
      f = 2.0 * y/x;
      return(f);
}
/*  End of subroutine func() */
/* ----------------------------------------------- */
```

## Program 29 RUNGE4

```
/ * ---------------------------------------------- *
 *  Main program                                  *
 *    This program computes the solution of first order   *
 *    differential equation of type  y' = f(x,y)  using   *
 *    the 4th order Runge-Kutta method            *
 * ---------------------------------------------- *
```

```
  *  Functions invoked                                          *
  *    Nil                                                      *
  * ----------------------------------------------------------- *
  *  Subroutines used                                           *
  *    func()                                                   *
  * ----------------------------------------------------------- *
  *  Variables used                                             *
  *      x - Initial value of independent variable              *
  *      y - Initial value of dependent variable                *
  *     xp - Point of solution                                  *
  *      h - Step-size                                          *
  *      n - Number of steps                                    *
  * ----------------------------------------------------------- *
  * Constants used                                              *
  *    NIL                                                      *
  * ----------------------------------------------------------- */

main( )
{
      int i,n;
      float x,y,xp,h,m1,m2,m3,m4;
      float func(float, float);

      printf("\n   SOLUTION BY 4th ORDER RK METHOD\n\n");

/* Input initial data */
      printf("Input initial values of x and y \n");
      scanf("%f %f", &x,&y);
      printf("Input x at which y is required \n");
      scanf("%f", &xp);
      printf("Input step-size, h \n");
      scanf("%f", &h);

/* Compute number of steps required */
      n = (int)((xp-x)/h+0.5);

/* Compute y at each step */
      printf("\n");
      printf(" -----------------------------------\n");
      printf("   STEP       X          Y          \n");
      printf(" -----------------------------------\n");
      for(i=1;i<=n;i++)
      {
         m1 = func(x,y);
         m2 = func(x+0.5*h, y+0.5*m1*h);
         m3 = func(x+0.5*h, y+0.5*m2*h);
         m4 = func(x+h, y+m3*h);
```

```
          x = x+h;
          y = y + (m1+2.0*m2 + 2.0*m3 + m4) * h/6.0;
        printf("%5d %15.6f %15.6f \n", i,x,y);
      }
      printf(" -------------------------------- \n");
/* Write the final value of y */
      printf("\nValue of y at x = %f is %f \n", x,y);

}
/*  End of main() program */
/* ------------------------------------------------ */

/* Defining subroutine func() */
float func(float x, float y)
{
      float f;
      f = 2.0 * y/x;
      return(f);
}
/*  End of subroutine func() */
* ------------------------------------------------ *
```

## Program 30 MILSIM

```
/* ------------------------------------------------ *
  *  Main program                                   *
  *    This program solves the first order differential *
  *    equation y' = f(x,y) using Milne-Simpson method *
  * ------------------------------------------------ *
  *  Functions invoked                               *
  *    NIL                                            *
  * ------------------------------------------------ *
  *  Subroutines used                                *
  *    func()                                        *
  * ------------------------------------------------ *
  *  Variables used                                  *
  *    x(1) - Initial value of independent variable  *
  *    y(1) - Initial value of dependent variable    *
  *      xp - Point of solution                      *
  *       n - Number of steps                        *
  *       h - Step-size                              *
  *       x - Array of independent variable          *
  *       y - Array of dependent variable            *
  * ------------------------------------------------ *
  *  Constants used                                  *
  *    NIL                                           *
  * ------------------------------------------------ *
```

```
main( )
{
      int i,n;
      float h,x[15],y[15],xp,m1,m2,m3,m4,f2,f3,f4,f5;
      float func(float, float);
/* Reading initial values */
      printf("Input initial values of x and y \n");
      scanf("%f %f", &x[1],&y[1]);
      printf("Input value of x at which y is required \n");
      scanf("%f", &xp);
      printf("Input step-size, h \n");
      scanf("%f", &h);
/* Compute number of computation required */
      n = (int)((xp-x[1])/h+0.5);
/* We require four starting points for Milne-Simpson
method.
Initial
    values form the first point. Remaining three points are
obtained
    using 4th order RK method */
      printf("\nINITIAL VALUES ARE %10.6f %10.6f", x[1],y[1]);
/* Computing three more points by RK method */
      printf("\n\nTHREE VALUES BY RK METHOD \n");
      for(i=1;i<=3;i++)
      {
        m1 = func(x[i],y[i]);
        m2 = func(x[i]+0.5*h, y[i]+0.5*m1*h);
        m3 = func(x[i]+0.5*h, y[i]+0.5*m2*h);
        m4 = func(x[i]+h, y[i]+m3*h);
        x[i+1] = x[i] + h;
        y[i+1] = y[i] + (m1+2.0*m2 + 2.0*m3 + m4)*h/6.0;
        printf("\n%5d %10.6f %10.6f", i, x[i+1],y[i+1]);
      }
/* Computing values by Milne-Simpson method */
      printf("\n\nVALUES OBTAINED BY MILNE-SIMPSON METHOD \n");
      for(i=4;i<=n;i++)
      {
        f2 = func(x[i-2], y[i-2]);
        f3 = func(x[i-1], y[i-1]);
        f4 = func(x[i], y[i]);
        /* Predicted value of y (by Milne's formula)*/
           y[i+1] = y[i-3] + 4.0*h/3.0 * (2.0*f2-f3+2.0*f4);
        x[i+1] = x[i] + h;
        f5 = func(x[i+1], y[i+1]);
        /* Corrected value of y (by Simpson's formula) */
        y[i+1] = y[i-1]+h/3.0 * (f3+4.0*f4+f5);
        printf("\n%5d %10.6f %10.6f", i, x[i+1], y[i+1]);
      }
      printf("\n\nValue of y at x = %f is %f", x[n+1], y[n+1]);
}
/*  End of main() program */
/* ---------------------------------------------------- * /
/* Defining subroutine func()*/
float func(float x, float y)
{
      float f;
      f = 2.0 * y/x;
      return(f);
}
/* End of subroutine func() */
/* ---------------------------------------------------- * /
```

APPENDIX E

# Bibliography

## E.1 NUMERICAL COMPUTING

1. Antia, H M, *Numerical Methods for Scientists and Engineers*, Tata McGraw-Hill Publishing Company, New Delhi, 1991.
2. Atkinson, L V, P J Harley and J D Hudson, *Numerical Methods with FORTRAN 77*, Addison-Wesley Publishing Company, 1989.
3. Buchanan, J L and P R Turner, *Numerical Methods and Analysis*, McGraw-Hill Book Company, 1992.
4. Chapra, S C and R P Canale, *Numerical Methods for Engineers*, 2nd Edition, McGraw-Hill Book Company, 1989.
5. Constantinides, A, *Applied Numerical Methods with Personal Computers*, McGraw-Hill Book Company, 1987.
6. Conte, S D and Carl de Boor, *Elementary Numerical Analysis*, 3rd Edition, McGraw-Hill Book Company, 1981.
7. Gerald, C F and P O Wheatly, *Applied Numerical Analysis*, 5th Edition, Addison-Wesley Publishing Company, 1994.
8. Griffiths, D V and I M Smith, *Numerical Methods for Engineers*, Oxford University Press, 1991.
9. Ledermann, W (Chief Editor), *Handbook of Applied Mathematics* (Volume III), John Wiley & Sons, 1981.
10. Mathews, J H, *Numerical Methods for Mathematics, Science and Engineering*, 2nd Edition, Prentice-Hall of India, 1994.
11. Scheid, F, *Numerical Analysis* (Schaum's Series), McGraw-Hill Publishing Company, 1990.
12. Yakowitz, S and F Szidarovszky, *An Introduction to Numerical Computation*, 2nd Edition, Macmillan Publishing Company, 1990.

## E.2 PROGRAMMING

1. Balagurusamy, E, *FORTRAN for Beginners*, Tata McGraw-Hill Publishing Company, New Delhi, 1985.
2. Balagurusamy, E, *Programming in ANSI C*, 2nd Edition, Tata McGraw-Hill Publishing Company, New Delhi, 1989.
3. Chamberland, L, *FORTRAN 90-A Reference Guide*, Prentice-Hall, 1995.
4. Etter, D M, *Structured FORTRAN 77 for Engineers and Scientists*, 4th Edition, Benjamin/Cummings Publishing Company, 1993.
5. Kochan, S G, *Programming in C,* Hayden Book Company, 1983.
6. McCracken, D D and W I Salman, *Computing for Engineers and Scientists with FORTRAN 77*, 2nd Edition, John Wiley & Sons, 1988.
7. Nyhoff, L and S Leestma, *FORTRAN 77 and Numerical Methods for Engineers and Scientists*, Prentice-Hall, 1995.

# Index